HZ BOOKS 华章IT

云计算与虚拟化技术丛书

PEARSON vmware PRESS

DevOps for VMware Administrators

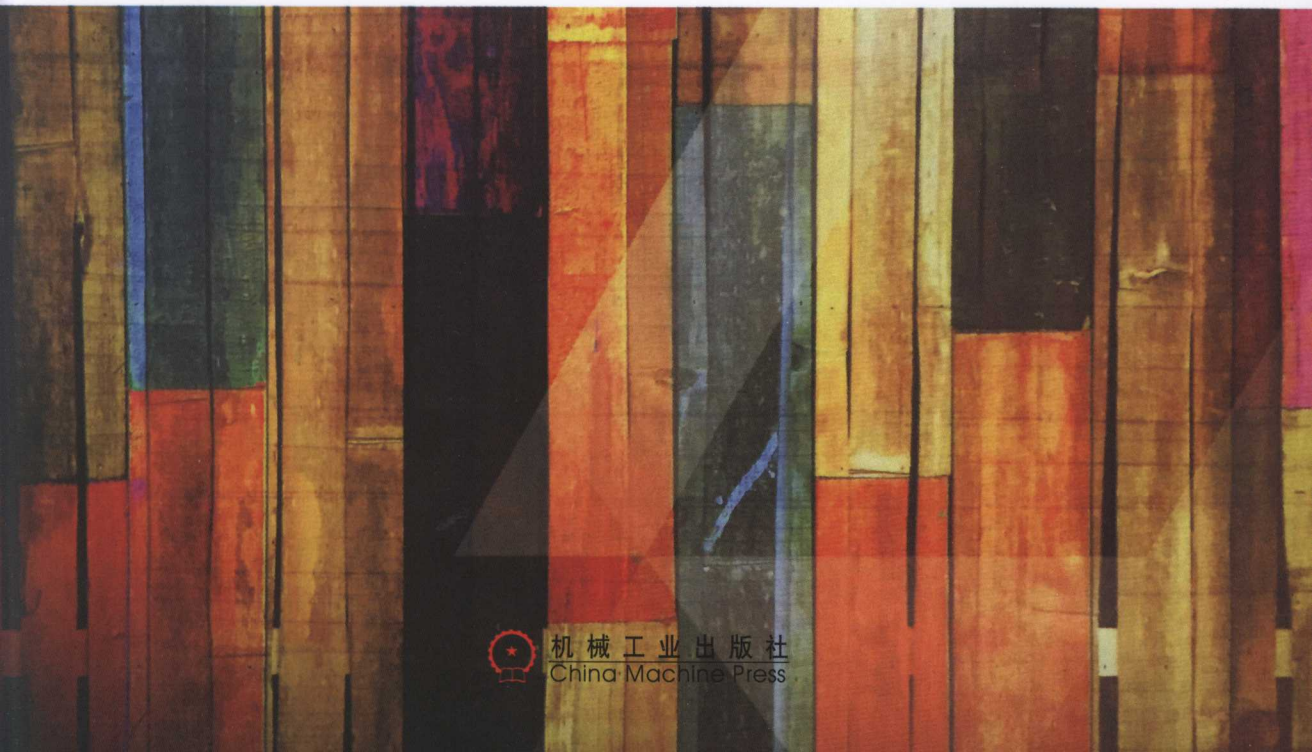
DevOps实战

VMware管理员运维方法、工具及最佳实践

[美] 小特雷弗 A. 罗伯茨（Trevor A. Roberts, Jr.） 乔希·阿特韦尔（Josh Atwell）
埃格勒·西格勒（Egle Sigler） 依弗·范·多恩（Yvo van Doorn） 著
姚军 译

VMware资深专家撰写，是第一本写给VMware管理员的DevOps权威指南

既系统介绍DevOps的基础概念和流行的工具，又详细讲解改变管理系统和交付服务的方法，涵盖DevOps环境配置、维护、编排、管理的各个环节，包含大量实例



机械工业出版社
China Machine Press

内容简介

本书由VMware资深技术专家撰写，是第一本写给VMware管理员的DevOps权威指南。书中既系统介绍了DevOps的基础概念和流行的工具，涵盖DevOps环境配置、维护、编排、管理的各个环节，又详细讲解改变管理系统和交付服务的方法，并且包含大量实例，可以帮助你快速了解并掌握DevOps工具、方法及最佳实践。

全书共19章，第1章讨论DevOps的概念；第2章介绍DevOps从业人员的一些流行工具；第3章介绍测试环境的建立；第4~6章介绍Puppet配置管理解决方案；第7~9章介绍Chef配置管理解决方案；第10章和第11章介绍Ansible配置管理和编排解决方案，第12章介绍Powershell预期状态配置；第13章探索VMware管理员在其环境中实施PowerShell DSC的方法；第14章讨论Linux容器的使用；第15章进一步讨论Linux容器，介绍Google Kubernetes；第16章描述如何安装、配置和使用Razor；第17章介绍Elasticsearch、Logstash和Kibana（ELK）栈；第18章介绍用于持续集成的Jenkins，讨论在代码提交到源代码库之后如何自动交付；第19章讨论VMware自身的DevOps倡议。

云计算与虚拟化技术丛书

DevOps for VMware Administrators

DevOps实战

VMware管理员运维方法、工具及最佳实践

[美] 小特雷弗 A. 罗伯茨（Trevor A. Roberts, Jr.） 乔希·阿特韦尔（Josh Atwell）
埃格勒·西格勒（Egle Sigler）依弗·范·多恩（Yvo van Doorn） 著
姚军 译

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

DevOps 实战：VMware 管理员运维方法、工具及最佳实践 /（美）罗伯茨（Roberts, T. A.）等著；姚军译 . —北京：机械工业出版社，2016.1
（云计算与虚拟化技术丛书）
书名原文：DevOps for VMware Administrators

ISBN 978-7-111-52478-6

I. D… II. ①罗… ②姚… III. 虚拟处理机 IV. TP338

中国版本图书馆 CIP 数据核字（2015）第 303916 号

本书版权登记号：图字：01-2015-5789

DevOps 实战
VMware 管理员运维方法、工具及最佳实践

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：关　敏　　责任校对：董纪丽
印　　刷：北京文昌阁彩色印刷有限责任公司　　版　　次：2016 年 1 月第 1 版第 1 次印刷
开　　本：186mm × 240mm　1/16　　印　　张：17.25
书　　号：ISBN 978-7-111-52478-6　　定　　价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzit@hzbook.com

The Translator's Words 译 者 序

互联网已经成为世界经济的焦点区域，越来越多的 Web 服务成为企业日常经营活动的核心，这对 IT 运营和开发都形成了巨大的挑战。

在传统的瀑布方法中，开发各阶段与运营之间是弱耦合的，开发阶段搜集的需求几乎不考虑运营，这也决定了在运营阶段发现的问题将成为久拖不决的痼疾，而且，传统方法无法适应 Web 服务频繁的更新换代。在此背景之下，以敏捷技术为代表的新型开发方法大行其道，同时，由于运营在业务中的地位越来越突出，IT 与运营之间的“竖井”必须打破，于是 DevOps 应运而生。

DevOps 的核心思想之一便是 IT 与运营团队的相互融合，运营人员参与开发全过程，指导开发人员生成运营友好的应用程序；开发人员参与运营环节，了解运营需求并及时消除代码中不利于运营的缺陷。在这种指导思想下，实现了开发环境、测试环境和生产环境的高相似度，可编程、自动化的全生命期配置维护管理，从而适应现代 Web 环境的高变动性以及高可用性、高可靠性要求。

虚拟化技术是现代云环境的基石，由于基础设施在很大程度上实现了软件化，自动化配置维护比以前更便于实现，而且硬件设施的安装、配置和维护也变成软件代码的一部分，可以像开发产品一样置于版本控制之下，因此，虚拟化技术与 DevOps 的有机整合成为令人期待的趋势。

本书由 VMware 的资深专家编著，系统介绍了 DevOps 的基础概念和流行的工具，这些工具包括最流行的第三方工具（如 Vagrant、Chef、Ansible、Razor、Docker、Microsoft PowerShell 等）和 VMware 自身提供的持续集成、交付和部署产品（如 VMware vRealize Automation），涵盖了 DevOps 环境配置、维护、编排、管理的各个环节，书中介绍的大量例子可以帮助读者快速了解工具的概念、使用以及如何与 VMware 虚拟化环境结合。对于想要试水 DevOps 的读者来说，这是一本不可多得的参考书。DevOps 是一个快速变化的领域，有

了这本入门书籍，读者有望跟上它的脚步，并在实践中不断提高，建设属于自己的理想世界。在翻译过程中，我们也深深感受到短短十年之间技术和理念变化带来的震撼。由于译者水平所限，书中难免存在不足，希望读者谅解并提出宝贵意见。

本书的翻译工作主要由姚军完成，徐锋、刘建林、陈志勇、宁懿、白龙、陈美娜、谢志雄、方翊等也为翻译工作做出了贡献，在此衷心感谢机械工业出版社的编辑关敏老师和其他有关人员为本书所提的宝贵意见。

译者

Preface 前　言

什么是 DevOps？是可以从供应商那里买到，解决所有 IT 问题的产品吗？是分析师用来引起 CIO 注意的行业流行词吗？虽然 IT 社区对 DevOps 的介绍接近于大肆追捧，但那更多的是因为 DevOps 确实能够带来好处，而不仅仅只是行业的广告宣传。

DevOps 这一术语指的是一组帮助各种规模的组织更快地从 IT 投资中获得价值的方法、理念和工具。这个词的确切含义是什么？

想象一下，在你的组织中为了将软件项目从概念阶段、软件开发一直推进到生产部署，需要多少时间和过程？这个过程越长，IT 组织向整个公司展示价值所需的时间就越长。由于技术无处不在，客户期待 IT 服务的交付像移动应用商店那么容易。他们不愿意为了一项功能的实现而等待数年，对客户的要求反应迟钝的公司难以获得长期的成功。

DevOps 如何解决客户交付速度问题？例如，配置管理技术可以避免服务器配置漂移，加速在线购买新服务器处理客户请求快速增长的过程。持续集成可以确保自动化测试在开发者提交源代码时进行。这只是本书所讨论技术的两个例子。

网络规模 IT 组织（如 Etsy、Netflix 和 Amazon Web Services）被视为 DevOps 的典范。但是，Gene Kim 的 DevOps 企业峰会参与者的数量证明 DevOps 也能给传统 IT 组织带来价值。

所以，做好思想准备，DevOps 正在来临。好消息是，你可以为所在 IT 组织 DevOps 行动的成功做出贡献。本书的目标不仅是介绍 DevOps 的概要思路，还将提供 DevOps 工具和技术的实例。

关于本书

在我们的经验中，DevOps 的概念和工具可以显著地改进 IT 运营。虽然 Amazon 和 Rackspace 等大型 IT 组织已经在它们的环境中实施 DevOps 并取得成效，但是许多企业级 IT 组织对 DevOps 实践仍处于熟悉阶段。

本书的目标是为读者提供上述 IT 组织获得成功所借助的 DevOps 工具的实操示例。

本书的读者

本书是为具备 VMware vSphere 虚拟化管理程序（hypervisor）和 Linux 操作系统使用经验的系统管理员所写。我们将循序渐进地介绍 DevOps 从业者所使用软件解决方案的使用方法，每章都提供后续研究所需的额外资源。

本书内容

本书介绍的主题从虚拟化专业人士如何获得 DevOps 实践知识的概述开始，然后讨论 DevOps 从业人员使用的各种工具。

第 1 章讨论 DevOps 的概念，包括这一术语的定义以及 DevOps 相关实践有助于 IT 组织成功的原因。

第 2 章介绍 DevOps 从业人员使用的一些流行工具。第 3 章准备建立测试环境，以使用本书中的示例代码。

第 4 ~ 6 章介绍 Puppet 配置管理解决方案，包括简介、多层次应用部署，以及 Puppet 与 VMware vSphere 服务器和虚拟机管理集成的介绍。

第 7 ~ 9 章介绍 Chef 配置管理解决方案，包括简介、常见系统管理任务，以及 Chef 和 VMware vSphere 环境管理集成的介绍。

第 10 章和第 11 章介绍 Ansible 配置管理和编排解决方案，包括这种技术和各种应用程序部署的基本知识。

第 12 章介绍 PowerShell 预期状态配置（PowerShell Desired State Configuration，DSC）的基础知识，包括 Microsoft Windows PowerShell 这一新功能的架构和主要用例。为了阐述 DSC 的基本功能、解释组成该功能的不同组件，提供了样板代码。

第 13 章探索 VMware 管理员在其环境中实施 PowerShell DSC 的方法。本章包括专门针对 VMware 管理员（可能不是 Windows 系统管理员）使用 DSC 提供额外价值及能力的用例。本章讨论了不同的方法，相应地强调和讨论了每种方法的建议和局限。

第 14 章讨论对企业 IT 组织来说相对新颖的一种应用程序部署范型：Linux 容器的使用。本章用实操示例讨论 Docker 容器管理系统的基础知识。

第 15 章进一步讨论 Linux 容器，介绍 Google Kubernetes，这是一种在数据中心大规模管理容器的开源工具。

第 16 章描述如何安装、配置和使用 Razor———一种全生命期自动配给工具，组合了安装、

服务器管理和配置工具。

第 16 章详细介绍 Razor 的所有关键概念和组件，首先描述 Razor 的工作原理和入门使用方法。一旦了解了 Razor 的概念和结合 DevOps 工具用于自动化配给的方法，你就能够发现 Razor 的不同功能组件。最后，本章介绍了 Razor 的最优安装和配置方法。

第 17 章介绍 Elasticsearch、Logstash 和 Kibana（ELK）栈。这些工具都可以单独使用，但是结合使用可以成为日志管理的完美组合。本章单独介绍每一种工具，以及如何组合它们、最大限度地利用它们的能力提升日志管理的效率。

第 18 章介绍用于持续集成的 Jenkins，讨论如何在代码提交到源代码库之后自动交付。

第 19 章讨论 VMware 自身的 DevOps 倡议，包括 VMware vRealize Automation 与 DevOps 工具的集成，以及新的 VMware vRealize Code Stream 解决方案。

致　谢 *Acknowledgements*

许多人对本书给予了帮助，我要感谢他们在任务完成中对我们的直接和间接影响：

感谢 Gene Kim 在忙于自己的著作（《The DevOps Handbook》）和 DevOps 企业峰会规划工作时抽出时间，指导本书的内容和写作过程的多个方面。

感谢 Nick Weaver 通过 Razor 方面的作品向 VMware 社区介绍 Puppet，开启了我的 DevOps 之旅。

感谢 VMware 出版社的 Joan Murray，他的有力支持推动了本书的写作。

感谢 Kelsey Hightower 在 Linux 容器及其大规模协调方面提供的专业知识。

感谢 Aaron Sweemer 提供了 VMware 内部的联络人，和本书的读者分享公司的 DevOps 愿景。

感谢我的合著者，感谢他们对我所领导的这一书籍项目的耐心和持续支持。

感谢 Scott Lowe、Nick Silkey 和 Matt Oswalt 为本书内容提供的宝贵反馈。

——Trevor Roberts，Jr.

我要感谢在写作我自己的那部分内容时给我提供帮助的几个人。感谢 Don Jones、Steven Murawski 和 Alan Renouf 在我寻求 VMware 管理员可能从 PowerShell DSC 得到的益处时提供的重要指导。没有他们的深刻见解和观点，我可能仍然在实验室中苦苦思索。还要感谢 Trevor Roberts，Jr. 邀请我参加这个项目。最后，我要感谢 VMware 社区的大力支持和对本书的兴趣。希望你们和我一样喜欢这本书。

——Josh Atwell

感谢开源社区，没有你们，我就不能拥有这么出色和令人惊异的工具。

——Egle Sigler

首先，我要感谢 Trevor Roberts，Jr. 给我参与本书创作的机会。感谢《 Promise Theory: Principles and Applications 》的合著者 Mark Burgess，Mark 在书中介绍了当今配置管理背后的科学知识，其中许多都是我们日常使用的。最后，我对 Chef 的每个人都心存感激，在 Chef 社区中，我才能迸发出许多灵感。

——Yvo van Doorn

关于作者 *About the Authors*

Trevor Roberts，Jr. 是 VMware 公司的高级技术市场经理。Trevor 拥有 CCIE 数据中心认证，是 VMware 数据中心设计和管理集中化认证高级专家。业余时间，Trevor 在 http://www.VMTrooper.com 通过 vBrownBag Professional OpenStack 和 Professional VMware 播客以及 Twitter（@VMTrooper）分享对数据中心技术的认识。他对 IT 社区的贡献得到公认，被授予 VMware vExpert、Cisco Data Center Champion 和 EMC Elect 的称号。

Josh Atwell 是 SolidFire 的云架构师，专注于 VMware 和自动化解决方案。10 年多的努力使他可以用少量代码通过各种自动化工具来完成自己的工作。Josh 已经有了两个儿子，2015 年年初，他和妻子 Stephanie 又生了一个女儿。他住在北卡罗来纳州的罗利，享受着和家人在一起的时间，他还喜欢高尔夫、有声读物和新的波本威士忌。Josh 是虚拟化社区的活跃分子，是 CIPTUG、VMUG 和 UCS 等技术用户组的领导人，而且还和其他人一起合作，准备通过 vBrownBag 播客和虚拟设计大师竞赛追求专业上的发展。Josh 还经常发表公开演讲，是 Mastering vSphrer 系列丛书的作者。他从不吝啬发表意见，在 vtesseract.com 上撰写博客，在 Twitter（@Josh_Atwell）上也是三句话不离本行。

Egle Sigler（@eglute, anystacker.com）现为 Rackspace 的首席架构师。她在职业生涯初期是一位软件开发人员，至今仍有着所有编写、测试和部署代码的人所具有的弱点，因为她有机会从事所有这类工作。Egle 的梦想是有朝一日，编写、测试和部署代码将成为无缝、轻松的过程，完全没有缺陷和挫折。Egle 坚信，知识应该共享，并通过撰写本书、发表讲话和会议上的探讨以及博客努力实践。

Yvo van Doorn 有 10 多年的系统管理经验。在职业生涯初期，他人工构建和配置“裸”服务器。在同辈人中，Yvo 成为配置管理和虚拟化的冠军。加入 Chef 之前，他在将西雅图一家小型技术公司的整个生产系统迁移到虚拟化平台时亲身见证了 VMware 产品的威力。他坚信 DevOps 所带来的文化变迁。在不忙于传播 Chef 的福音时，他可能会享受醉人的 IPA 啤酒，探索好的户外运动或者继承自己的荷兰传统，一边吃着高达干酪，一边观看橙衣军团丢掉世界杯。Yvo 和妻子及黑色的赖伯犬一起住在华盛顿州西雅图市。

About the Reviewers 关于评审人员

Scott Lowe，VCDX 39，在 IT 行业工作了 20 多年，目前是 VMware 的工程架构师，专注于网络虚拟化、开源和云计算的汇聚点。他还花时间参与了许多 DevOps 相关的产品和项目。

Randall "Nick" F. Silkey，Jr. 是 The Rackspace Cloud 的高级系统工程师。他热心于基础架构自动化和发行工程，喜欢组织得克萨斯州奥斯汀的几个专业技术组织。Nick 还在当地及全国性会议上发表关于持续集成及运营工程的演讲。工作之余，Nick 享受着和妻子 Wendy 及 3 个孩子在一起的时光。

Matt Oswalt 是一位全能的技术迷，目前主要关注在联网工作中引入自动化工具和方法。他进入 IT 界时在一家大型零售连锁企业担任应用程序开发人员。之后，他担任网络基础设施方面的顾问达 4 年之久。现在，他将这两方面的技能结合起来，建立更灵活、有弹性的网络基础设施。他常常参加自动化和 DevOps 社区，帮助推动网络自动化和 SDN 方面的交流。他经常在 keepingitclassless.net 上的个人博客以及 Twitter（@Mierdin）发表关于这一领域及传统基础设施的作品。

关于贡献者 *About the Contributing Author*

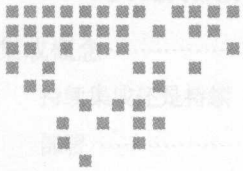
Chris Sexsmith 是本书的贡献者。Chris 在过去 4 年内担任 VMware 全球卓越中心的员工解决方案架构师，主要专注于自动化、DevOps 和云管理技术。Chris 住在加拿大不列颠哥伦比亚省的温哥华，在那里攻读 MBA，并尽可能地将其余时间用于观看冰球比赛。Chris 和他的团队领导 LiVefire 项目，专注于软件定义数据中心（SDDC）内专家和合作伙伴解决方案的实现。

Contents 目　录

第一部分 *Part 1*

DevOps 概述

- 第 1 章　DevOps 简介
- 第 2 章　DevOps 工具
- 第 3 章　建立 DevOps 配置管理测试环境



Chapter 1 第 1 章

DevOps 简介

DevOps 是一种系统部署方法学，组织可以用它来改善项目部署的速度和质量。它不仅仅只是另一个流行词，传统的 IT 组织正在认真研究 DevOps 实践和工具对它们实现目标带来的帮助，这就是一个明证。DevOps 只对云规模组织（如 Netflix 和 PayPal）有好处吗？当然不是，DevOps 实践对任何规模的组织都有好处。

本章包含如下主题：

- DevOps 原则概述
- DevOps 原则在组织中的实际运用
- 磨练 DevOps 知识和技能的重要性

1.1 DevOps 原则概述

DevOps 包含组织互动和部署工具及实践的变化，主要强调识别和缓解生产率瓶颈。你们当中的一些人可能阅读过 Gene Kim 所著的《The Phoenix Project》，书中他将 DevOps 的重要原则归结为 DevOps 的三条道路：

- 第一条道路：优化从开发到 IT 运营的工作流。
- 第二条道路：缩短和放大反馈循环。
- 第三条道路：鼓励试验，快速从故障中学习。

这些原则和 John Willis 及其他 DevOps 思想领袖所讨论的流行概念 CAMS（文化、自动化、计量和共享）相符：

- 文化：故障不应该立即引起过失认定，改变团队成员对部署方法和故障响应的思维方式。

- 自动化：人工方法容易招致故障。使用能够以可靠的方式重复、快速部署环境的工具。
- 计量：监控和分析对于成功必不可少，否则，故障的根源分析就不可能实现。
- 共享：个人 / 团队独占信息，以维持地位提升或者团队依赖性的“摇滚明星”心态在 IT 文化中站不住脚。这种心态不会加快生产速度，而会降低生产速度。

这些想法中，有些可能需要管理层的权力才能推动变化。那么，它们对于作为 IT 部门中虚拟化专家的读者来说意味着什么？你如何帮助引导这种变化？

虚拟化专家处于一个独特的地位，能够帮助开发团队和运营团队保持一致。在融合式基础设施的世界中，我们在规划和部署虚拟基础设施时已经熟悉了多学科（计算、网络、存储、安全等）的协调。这种专业知识对于协调组织中的开发和 IT 运营团队至关重要。

不管我们部署虚拟基础设施有多快，企业项目的成功将取决于从开发迁移到生产的效率。

想象一场接力赛：如果团队没有预先准备（回顾旧的影片、训练交接棒、力量训练等），在赛场上就肯定会犯错（掉棒、选手相互绊倒等）。IT 运营将会为误期而受到责备，“影子 IT”的使用将更加盛行。

在前一段中我强调“预先”，是因为当开发团队完成敏捷发行过程时，再启动部署工作往往已经太迟了。如果你的一位或者多位团队成员成为开发组织的顾问，在整个“冲刺”过程中都和他们在一起工作，而不仅仅在发行的末期，会怎么样呢？我们将以下面讨论的一个虚构项目作为背景，说明这些方法。

例如，假定你的公司 DevWidgets 决定发行一个软件即服务（SaaS）解决方案，包含其集成开发环境（IDE）和源代码管理（SCM）软件解决方案（Taao 项目）。目标是利用互联网，通过在线结对编程、代码评审网络会议等手段，培养地理上分散的团队成员之间的互动。

Taao 项目是高级领导层确认为公司旗舰产品的重大举措。有些高管人员曾经阅读过《The Phoenix Project》，他们希望对该项目实施“DevOps 工作”，因为他们对过去的发行安排感到不满。CEO 甚至考虑建立专门的 DevOps 团队，全部配属新雇用的人员（因为人数越多，项目总是推进得越快，对吗？）。

开发和 IT 组织的经理们努力使高级领导层保持镇静，请求领导给出时间，并承诺在当月月末提交一项计划。在这方面你能提供什么帮助？下面几节我们将探索你可能采取的措施。

1.2　采用系统思维

系统思维意味着将涉及软件发行版本部署的所有团队当成一个紧密相连的单位，而不是日程安排相互冲突的多个分散团队。这些团队包括信息安全、运营、开发、质量保证（QA）、产品管理等。

在我们的讨论中所提到的系统指的是经历整个软件开发生命期（规划、开发、QA、部署和维护）的任何项目，不管这些项目是供内部还是外部消费的。

1.2.1 改变团队的互动方式

我们将焦点放在第一件应该做的事：成为开发团队的顾问。和负责定期规划会议的开发团队领导（在敏捷的术语中称为产品负责人和敏捷教练）对话，要求加入他们的一些回忆。主动倾听可能需要基础设施采购 / 部署的任何团队目标或者可交付成果。下面的几段强调来自开发团队的需求 / 规格示例，说明了在这种会议期间你所能提供的反馈 / 专业意见。

- “我们需要用于客户源代码的长期数据存储。”

开发团队不一定直接说出他们需要一个新的磁盘阵列，甚至没有意识到需要采购存储设备。但是，对存储系统的认识会提示你，有必要和存储主题专家（SME）一起进行容量规划。

你还需要在开发人员讨论用户数据生命期管理时考虑不同的存储层次。

- “我们需要在生产环境中运行独立于客户使用实例的 Taao 项目实例。”

这立即让你想起附加虚拟网络或者虚拟防火墙的需求，这能够使生产和客户数据流量与内部流量分隔开。你还需要咨询信息安全团队，了解提供客户数据分离保证的审计过程。

- “在我们的最后一个发行周期中，代码在我的便携电脑上成功编译和运行，但是在部署到生产环境时崩溃。”

开发人员可能提出一个在之前的代码部署中遇到的问题，提交的代码在她的便携电脑上工作正常，但是在生产部署时发生故障。她的本地开发环境运行的是 Ubuntu 虚拟机（VM），和非军事区（DMZ）中的 CentOS 服务器上实施的安全补丁或者防火墙规则不匹配。所以，需要立即打补丁以满足最后限期（对于开发团队和运营团队都是一个漫长的夜晚）。你对 Vagrant 和 Packer 等环境构建工具和 Puppet、Chef 及 Ansible 等配置管理工具的认识，使你能够构建一个用于开发人员便携电脑且匹配生产服务器规格的标准测试环境。

由于开发团队领导了解了运营团队在发行周期早期为了缓解开发环境问题所做出的努力，你在协调与此努力相关的一些活动时就会更容易一些。首先，这意味着开发团队需要确保其成员不会将任务推给其他团队。（也就是说，“现在是周五下午 5 点了，我不知道代码能不能在生产环境中正常工作，但是无论如何，我都会把它提交到存储库！”）

某组织在最近的 PuppetConf 上发表讲话，分享其代码部署策略：当代码部署到生产环境时，编写代码的开发人员参与更改窗口期的轮班。这样，如果代码中出现任何复杂现象，他们可以立即协助运营团队查错和解决。坦率地说，这也使开发人员有动力编写更好的代码，避免在凌晨 3 点接到电话去修复缺陷。

在继续之前，对于可能阅读本书的经理，Jez Humble 强烈建议：不要构建新的 DevOps 团队！雇用在 DevOps 方法学上有专业能力的开发人员、QA 工程师、运营人员是个好主意，

但是创建不同于组织其余团队的全新 DevOps 团队只会带来新的“竖井”，可能对你的目标造成反作用。相反，应该像前面提到的那样，继续更好地协调各个团队，使他们像一个团队那样思考和行动，而不是各自寻求自己的最大利益。

1.2.2　改变基础设施部署方法

数据中心的一些扰人而又常见的现象可能妨碍系统的进展：手工制作的“金映像”、雪花服务器和易碎箱。

服务器（不管是物理还是虚拟服务器）部署的常用方法是维护一组包含必要更新、补丁、设置等内容的操作系统配置，人工运用这些配置使系统立刻为部署做好准备。这些配置被称作“金映像”，传统上采用 ISO 形式，已经根据某个运行手册人工应用补丁。最近，金映像已经采用模板 VM 的形式保存。但是，老实说：金映像也可能生锈！

金映像本身没有什么问题。但是，构建它们的方式可能带来问题。如果人工构建，该过程可能是一个全职工作，必须跟踪模块更新、安全补丁等，然后重新配置 ISO/ 模板 VM 供以后使用。必须有一种更有效的方法，也的确有！

使用第 4 ~ 13 章介绍的配置管理技术，可以自动构建映像。随着服务器配置更改并登记到 Git（第 3 章中介绍）等存储库，Jenkins（参见第 18 章）等持续集成（CI）系统可以输出一个更新的金映像。CI 系统可以使用 Packer 等工具生成用于 Vagrant、虚拟化管理器和云提供商的模板 VM。

Martin Fowler 提出了“雪花服务器”的思路，雪花服务器是一台物理机器或者 VM，最初是为了保持标准化的良好愿望。但是，不管是为了完成故障报告而进行快速修复、高管要求的特殊项目还是其他原因，这种服务器都变成高度定制的。这些特殊更改解决了短期问题，但是会造成长期的麻烦。当对这些服务器上的软件包升级时会发生什么？安全补丁甚至操作系统升级又会发生什么？

如何解决雪花服务器的问题？首先，我们通过登录到服务器命令行界面强制避免任何更改；不允许一次性执行前端和软件包更新工具或者更改配置文件。所有服务器更改都只能通过配置管理技术进行。

对于现有的雪花服务器，必须进行一些乏味而不必要的工作。首先，检查服务器配置，以便准确地知道需要修改的配置文件、需要安装的软件包和其他需要包含在配置管理脚本中的关键设置。多次迭代构建和测试服务器，每次都对配置管理脚本进行调整，直到测试环境与雪花服务器完全相同。将更改提交到源代码库，就可以拥有一个可在生产服务器无法恢复时使用的可重复构建过程。

易碎箱这一术语来源于 Nassim Nicholas Taleb 的《 Antifragile 》一书，描述了一台运行关键软件栈的物理或者虚拟服务器，每个人都害怕接触它，因为业务可能因此遭遇重大故障。通过许多支持电话 / 专业服务，这台服务器已经达到了一个稳定状态，糟糕的是，管理该服务器的 SME 离开了公司。

如果稳定性确实是个考虑因素，进行物理 - 虚拟转换或者虚拟 - 虚拟迁移到 VMware vSphere 平台是个好主意。这样，你可以利用分布式资源调度器（DRS）、存储 DRS、高可用性（HA）等 VMware 基础设施可靠性机制。在服务器转换且拥有成功的业务持续性工作流（备份、快照）后，就可以考虑是否可以利用前面雪花服务器的弥补措施复制这一服务器。开发和 QA 工程师就可以很容易地构建一个测试环境而不接触易碎箱。

1.2.3 改变软件开发和部署方法

我们已经解决了基础设施部署的速度和质量问题，那么团队如何减少从开发、展示到生产阶段代码移动引起的缺陷？众所周知，在开发周期中越早发现缺陷，修复的代价就越低。这就是我们首先要有 QA 团队的原因。

但是，如果我们可以在代码移交给 QA 工程师之前就捕捉到缺陷，又会如何？这就是前面提到的持续集成系统的用途所在。我们后面会更详细地讨论 CI，基本要点是，当你将源代码提交到存储库时，CI 系统可以修改并设置为自动执行对提交代码的一系列单元测试。在过程结束时，可以构建一个软件包并自动分发给 QA 团队，实施他们的全面测试。

1.2.4 经常收集和响应有用的系统反馈并相应调整

系统思维的转变只有在有能力监控和分析系统性能时才能成功。业务的变化节奏似乎是指数级的，消费者对系统响应能力和正常运行时间有更高的预期，被动的问题解决方案不再成为选择；相反，你的团队必须在问题发生之前预测到它们，以维护系统稳定性。

日志分析可能是最重要的工具之一，尤其是在启用了二进制代码的调试选项时。如果你的环境由较多的服务器组成，就应该采用某种形式的自动化日志分析。这方面的选择很多，包括 VMware vRealize Log Insight、Splunk 甚至 Logstash、Elasticsearch 和 Kibana 等开源工具（第 17 章中介绍）。这些解决方案使系统管理员能够检查重复的事件与低下的性能关联。当团队更加熟悉这些工具，且更擅长识别问题时，系统的实用性就会提升。

1.3 增进 DevOps 知识和技能

一旦在系统思维和改进的系统验证上构建了很好的基础，团队对定期试验新功能就会更加自信。DevOps 实践使开发人员能够分阶段逐步地投产各种功能。这样做的好处之一是，开发人员可以利用针对选择的客户的限定发行版本对新功能进行 Beta 测试，收集基础设施影响和用户接受度方面的指标。有效的日志分析方法能够在重大问题蔓延之前发现它们。

持续的学习和改进不只是用于新产品功能的部署：持续发展自己的技能、扩展知识集合也很重要。除了跟上 VMware 技术的最新、最杰出的发展之外，还要积极学习有利于和开发人员进行更有意义互动的新技能。学习公司使用的编程语言的基础知识，培养对敏捷过程的

兴趣。和公司开发人员的沟通越有效，项目从开发到生产的迁移就越顺畅。

积极增进对“快速流程”的理解，Gene Kim 用这个术语描述合理的产品开发和部署。这方面的最佳方法往往是参加有关 DevOps 实践和工具的聚会、会议、用户组（例如，DevOpsDays、DevOps 企业峰会、PuppetConf 和 Chef Conf）。阅读来自思想领袖、关于这一主题的数据 / 博客帖子 / 推文，并在行业活动和社会化媒体上与其交流。

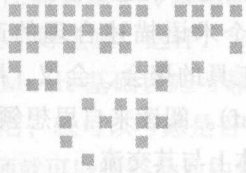
1.4　小结

我们已经确定了 DevOps 的概念和对组织的好处，下面我们将更仔细地研究有助于团队成功的一些工具。

参考文献

[1] *The Phoenix Project*, by Gene Kim, Kevin Behr, and George Spafford

[2] “What Devops Means to Me,” by John Willis: https://www.getchef.com/blog/2010/07/16/what-devops-means-to-me/



Chapter 2 第 2 章

DevOps 工具

有一些工具能够帮助团队采用 DevOps 技术。本章介绍这些工具，在本书后面，将用实操示例更详细地介绍。

本章包含如下主题：

- 为成功而组织
- 服务器部署
- 配置管理
- 持续集成
- 日志分析

2.1 为成功而组织：看板

传统的运营团队任务管理通常涉及某种单据系统。故障单据很适合于跟踪问题，并在问题解决以后进行历史分析。故障单据系统的问题在于，它们给应该相关的任务之间的关联带来了困难。另一个挑战是生产交付瓶颈根源的识别。而且，工作者们可能无法看到其他人处理的故障单据，从而无法借助他人的专业知识。最后，管理工作过程、避免过度分配运营团队成员的最佳途径是什么？也就是说，如果运营团队总是专注于堆积如山的指派任务，他们何时才有时间改善系统，偿还技术债务呢？我们如何正确排定工作的优先级，考虑任务之间的依赖性？

看板（Kanban，字面翻译为“标记卡片”）系统有助于解决这些问题，以及其他的一些问题。这种方法是 Taiichi Ohno 在开发丰田制造系统时为了实现即时生产（JIT）目标而开发

的，它通过检查制造过程不同步骤的流程，识别需要补救的瓶颈，使系统更加高效。具体的思路是，缓解瓶颈，就会将工作任务从在途状态带到完成状态。限制在途工作可以为工作者带来空闲时间，对制造过程进行改进（例如，在缓解旧瓶颈的同时识别和消除新瓶颈）。图2-1展示了制造过程的估计完成时间没有实现的例子。

随着工作流从原材料输入到制造过程，最后进入装配过程，我们可以看到估算的完成时间和实际不符。装配过程似乎有某些问题。在进一步调查中，工厂工人们可能发现在金属薄板上喷涂特种涂料的过程效率不高。例如，服务器面板制造过程的输入可能表现出没有正确应用涂料的信号，所以进行返工，导致整个过程的效率不高。如果喷涂过程中固定薄钢板的托架没有超载，输出产品的制造可能从一开始就是正常的，不需要太多的返工。

在前一个例子中，如果工人埋头修复没有正确喷涂的金属薄板，而没有去识别问题的真正根源，就会继续浪费精力。我们从来没有在IT运营工作流中看到这种问题，对吗？

你可能会觉得奇怪，我们为什么在关于DevOps工具的章节中讨论制造工程组织方法。但是，在成功地改变工作方式之前，我们必须用一种条理性的方法来安排工作、识别系统中的问题。

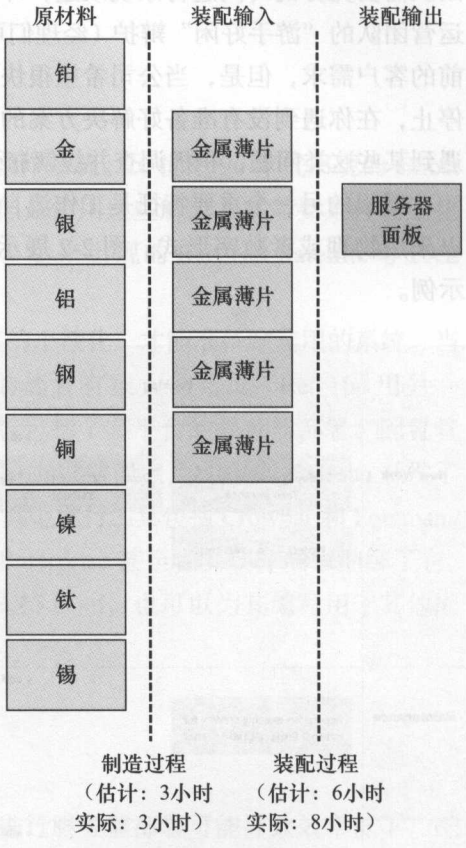


图2-1 制造过程

尽管看板是制造中的一个学科，但是通过David J. Anderson和其他人的努力，将精益的概念与丰田的看板方法相结合，在IT业流行起来。我们不对看板做全面介绍，但是将讨论有助于IT运营团队的关键点。

看板系统最重要的特征是工作过程管理。在新工作请求时分配所有IT运营人员会造成效率低下，这一点似乎有些违反直觉。但是，运营团队面对和制造团队类似的问题。例如，如果团队使用预先制作的ISO金映像构建服务器，如果金映像有一段时间没有更新，那么在需要人工更新应用程序和打补丁时就有可能出现错误。如果团队的管理者将重点主要放在满足服务器请求，以至于100%的工作人员都参与这类工作流，而没有寻找改善过程效率的方法，这样的情况还会持续。不过，人为的错误可能引起失败，重要的安全更新可能遗漏，容易遭到攻击的端口保持打开，必须加以补救才能确保服务器不会遭到攻击。团队可能习惯于这种低效的状态，管理层也对产量感到满意。但是，很多时间浪费在这种重复劳动上，这种状态被称作“技术负债”。

技术负债是在计划好的工作期间，由于错误或者效率低下造成的所有计划外工作。运营团队需要充分的时间进行系统改进，即使这一工作没有相关的具体可交付成果。我并不是为运营团队的“游手好闲”辩护（经理们可能这么想）。运营团队看上去似乎很高效地服务于当前的客户需求，但是，当公司希望很快增加运营规模时会发生什么情况？现有的工作流不能停止，在你遇到没有准备好解决方案的伸缩性问题时，就要对系统进行改进。如果现在已经遇到某些这类问题，积极调查并缓解瓶颈，可以帮助你增进效率。

看板的另一个重要特征是工作流自始至终的可视化。最流行的展示方式是看板图，它可以采用物理或者数字形式。图 2-2 展示了可供第 1 章介绍的 DevWidgets 公司使用的看板图示例。

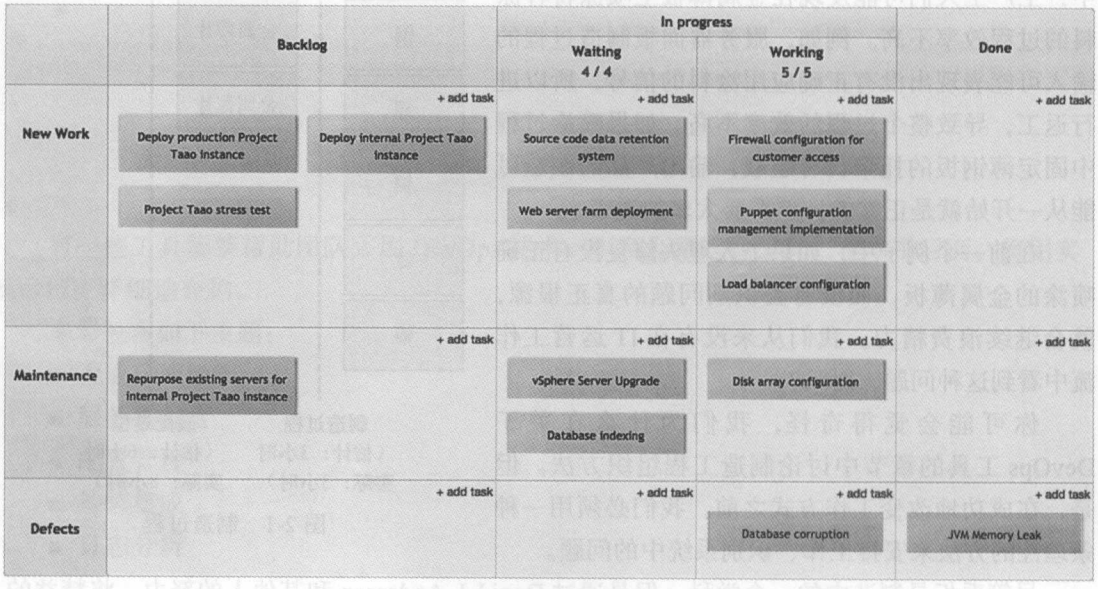


图 2-2 在线看板图

看板图的思路是每个任务由一张索引卡或者即时贴（也被称作看板）表示，在看板图左侧的“积压工作”（Backlog）分类下排队。“积压工作”和“完成”（Done）之间的栏目代表工作过程。在工作任务移出积压工作栏时，在它们上面放置工作过程（WIP）限制（例如，4/4 和 5/5）。在一张看板移到工作流的下一栏之前，不应该在 WIP 栏上放置其他工作。

图 2-2 展示了不同的任务行——也称为泳道。这些泳道对应于不同类型的工作（例如，新产品、维护和缺陷）。正确地为工作分类有助于优先级的排定。

团队第一次开始使用看板时，WIP 限制可以根据过去的经验设置。以后，随着瓶颈的缓解和团队效率的改进，这些限制可以更改，这样团队就可以随着时间的推移不断寻求工作流的改善。记住，目标是持续改善，同时避免团队 100% 投入新的任务。如果你对在团队中引入看板方法感兴趣，咨询该学科从业者中的佼佼者（如 Dominica DeGrandis 或精益看板大学

认可的从业者）是值得的。你的看板图不一定要和图2-2中的相似，目标是开发对团队有意义的清晰、实用系统。

2.2 服务器部署

系统管理员当然无法负担人工部署服务器场中的操作系统。20世纪90年代出现的Ghost和其他映像备份工具使我们离快速部署更近了一步，但是使用这些解决方案需要某种初始化过程（例如，Windows的Sysprep），新系统才能使用。后来出现的Red Hat Satellite等PXE启动实用程序加速了这种部署过程。

当今的工作流程需要在OS部署完成之后进行更多的个性化，才能准备好实用的系统。当然，VMware发布了Auto Deploy，对vSphere服务器的部署有很大的帮助。Red Hat用另一种眼光看待系统部署方法，开发了CloudForms，该系统包括了多平台的服务器部署、配置管理（可与Chef和Puppet集成）、服务器生命期管理等。在开源战线上，Michael DeHaan开发了Cobbler，Nick Weaver推出了Razor（第16章中介绍）。其他流行工具包括Crowbar和Foreman。

Razor已经变得越来越流行，最新版本的Puppet Enterprise甚至在安装介质中捆绑了它。Razor不只限于与Puppet协同工作，它的代理组件还支持Chef，也可以为其编写用于其他配置管理技术的插件。

2.3 配置管理

正如第1章中讨论的，只依靠手工制作的金映像进行服务器部署可能导致效率低下。配置管理技术（CM）可以显著地改善金映像构建和生产系统部署的速度及可靠性。当你把服务器配置当成软件看待，就可以利用Git（第3章中介绍）等源代码管理系统跟踪环境变化。

CM技术还可以用于配给与生产服务器配置完全相符的一致开发环境。这可以消除开发和运营团队之间“但是它在我的开发工作站上工作得很好”的争论。CM技术的设计很灵活，可以通过动态调整服务器特性（例如，OS风格和位置），在不同平台上使用相同的指令集。本书介绍的CM技术——Puppet（第4 ~ 6章）、Chef（第7 ~ 9章）和PowerShell DSC（第12 ~ 13章）——是描述性语言，你可以描述配给资源的预期状态，而不用担心工作是如何完成的。

配置管理的好处可以通过使用Ansible（第10 ~ 11章）、Fabric或者MCollective等编排系统大规模实现，这些系统用命令式的风格描述环境状态。编排框架允许配置管理在多个系统上以受控方式并行执行。

Ansible也可以视为一种CM技术，因为它能够描述预期状态。有些公司可能选择仅使用Ansible，而其他一些公司则使用Puppet/Chef的组合，由Ansible编排CM的执行。

2.4 持续集成

Jenkins（第 18 章中介绍）和类似的解决方案可能显著地节省开发和运营团队的时间。对于开发团队，如果编写了好的单元测试，它可以在代码移交给 QA 之前及早识别缺陷。对于运营团队，能够更加确保他们不会在预演阶段之前遇到未经验证的代码。

Jenkins 可以和 Git 及 Gerrit 集成，它们可以在开发团队将代码提交到软件存储库之后，立刻自动将代码提交给 Jenkins 供编辑和测试执行。而且，Jenkins 的作业历史和控制面板输出允许所有团队成员检查自动化测试和构建循环的各个阶段。

2.5 日志分析

系统的成功必须是可计量的。验证系统稳定性的最佳手段是什么？观察日志！你可能需要一支军队才能人工查看日志。而且，并不是整个团队都拥有日志通常需要的访问特权。幸好，整个运营团队可以得益于市场上的一些开源和商业化解决方案，这些方案可以用于日志收集和数据可视化。

VMware vRealize Log Insight 和 Splunk 是第一个想到的熟悉名称。还有 Logstash 等开源解决方案，它能够与 Elasticsearch 和 Kibana（第 17 章中介绍）、Graphite 以及 DevOps 社区中流行的其他工具相结合。因为这些开源工具都是 Linux 操作系统中包含的软件包，它们可以用你所喜欢的软件包管理器（如 yum 或者 apt）部署。

2.6 小结

现在，我们已经简单地概述了 DevOps 从业者可用的工具，第 3 章将聚焦于如何构建一个测试环境，以便跟踪本书其余部分介绍的工作流。

参考文献

[1] *Kanban*, by David Anderson
[2] *Toyota Kata*, by Mike Rother
[3] *The Phoenix Project*, by Gene Kim, Kevin Behr, and George Spafford

第 3 章 Chapter 3

建立 DevOps 配置管理测试环境

在不顾一切地深入 DevOps 工具的使用之前，我们先来研究如何建立测试环境，以便做好充分的准备。我们还要了解如何开始像对待开发人员的源代码那样处理基础设施指令。

本章包含如下主题：

- 用 AutoLab 进行环境配给
- 用 Vagrant 进行环境配给
- 用 Packer 创建映像
- 管理源代码
- Git 源代码控制

3.1 用 AutoLab 进行环境配给

AutoLab 是 Alastair Cooke 和 Nick Marshall 在 VMware 社区中其他人的帮助下开发的 vSphere 测试实验室配给系统。AutoLab 已经十分流行，至少有一个云提供商（Bare Metal Cloud）允许你在它的服务器上配给 AutoLab 安装。在本书写作时，AutoLab 支持最新版本的 vSphere，可以运行包含 VSAN 支持的虚拟实验室。如果不熟悉这个工具，可以在 http://www.labguides.com/ 首页上找到 AutoLab 链接。在填写注册表单后，就可以访问虚拟机（VM）模板，开始设置自己的 AutoLab。对于需要额外协助的人，该团队在 AutoLab YouTube 频道上有一组精选的帮助视频。

3.2 用 Vagrant 进行环境配给

Vagrant 是 Mitchell Hashimoto 创建，由他的公司 HashiCorp 支持的环境配给系统。它能够帮助你根据模板文件（称作 Vagrantfile）中定义的模式快速配置 VM。Vagrant 可以在 Windows、Linux 和 Mac OS X 操作系统上运行，支持流行的桌面虚拟化管理器（如 VMware Workstation Professional、VMware Fusion Professional 和 VirtualBox）。Rackspace 和 Amazon Web Services 等云提供商也可以用作你的测试环境。本书中的 Vagrant 示例基于 VMware Fusion 插件，但是我们提供的例子作少数修改就可以用于其他虚拟化管理器和云平台。如果按照本章中的例子学习，一定要为创建的每个 Vagrantfile 建立一个新目录。


注意 VMware Fusion 和 Workstation 提供商需要购买插件的许可证，更多细节参见 http://www.vagrantup.com/vmware。

安装 Vagrant 和 VMware Fusion 或 Workstation 插件后，你需要找到一个用于该系统的 Vagrant 盒子（box）。Vagrant 盒子可以看作一个 VM 模板：可以根据 Vagrantfile 中指定设置修改的预安装操作系统实例。Vagrant 盒子是所需操作系统的最小安装，有些盒子不超过 300MB。盒子的思路是提供一个完全由所选择的自动化工具配置的操作系统骨架。

有些盒子创建者选择包含 Vagrant 支持的流行配给系统（如 Puppet、Chef 等）的二进制代码，但是其他盒子创建者不这么做，用户必须使用命令行配给程序部署自己喜欢的配置管理解决方案才能使用。盒子的创建过程超出了本书的范围，但是，如果想要开发自己的盒子，可以尝试 veewee 和 Packer（本章后面将讨论）等工具。

在过去的 Vagrant 版本中，必须在初始化 vagrant 环境时指定所要使用的盒子文件的 URL：

```
vagrant init http://files.vagrantup.com/precise64_vmware.box
```

如果盒子文件在你的计算机上，可以用文件完整路径代替 URL。

从 Vagrant 版本 1.5 开始，HashiCorp 推出了 Atlas 系统（原称 Vagrant Cloud），这是一个在线存储库，如果使用保存在其站点上的账户和盒子名称，Vagrant 将在存储库中搜索盒子：

```
vagrant init hashicorp/precise64
```

对这两类语法都加以了解是很好的，因为对于没有寄存在 Atlas 上的任何盒子，都必须使用旧的方法。在线站点是搜索各种操作系统、代替自行构建的好地方。

vagrant init 命令自动创建一个简单的 Vagrantfile，引用你所指定的盒子。程序清单 3-1 展示了上述命令生成的默认 Vagrantfile。

程序清单 3-1 默认 Vagrantfile

```
# -*- mode: ruby -*-
# vi: set ft=ruby :
```

```
# Vagrantfile API/syntax version. Don't touch unless you know what you're
doing!
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "hashicorp/precise64"
end
```

注意，为了节约空间，我删除了初始化 Vagrantfile 时自动生成的注释。不过，如果查看注释，会看到使用配置管理技术在 VM 启动时自动化修改的有用提示。目前，配给 VM 的可用选项包括基本 shell 脚本和配置管理工具（如 Puppet、Chef 和 Ansible）。这样很有价值，因为开发和测试环境可以使用和生产部署完全相同的设置，从而消除出现部署问题时没完没了的“它在我的笔记本电脑上运行得很好”的讨论。Vegrant 还添加了 Docker 支持，配给系统可以自动安装 Docker 守护进程，下载指定使用的容器。

有了 Vagrantfile，现在可以使用如下命令启动第一个测试环境：

```
vagrant up --provider=vmware_fusion
```

注意　如果你在 Windows 或者 Linux 平台上使用 VMware Workstation，可以使用不同的提供者：vmware_workstation。

现在，可以用如下命令登录 VM：

```
vagrant ssh
```

查看 VM 中的 /vagrant 文件夹，将会看到它包含了你的 Vagrantfile。这是一个为 VM 自动创建的共享文件夹，可以轻松地在桌面上传入和传出文件，而不需要使用 SCP、FTP 等。

如果检查操作系统资源，会注意到有一个 vCPU 和 512MB 的 RAM。这对你想要运行的应用程序可能不足，所以，我们来看看如何修改分配给 Vagrant VM 的资源。

首先，我们删除这个 VM，以便转向其他的配置选项，退出 VM 并使用如下命令：

```
vagrant destroy
```

Vagrant 将要求确认是否真的要删除该 VM，也可以使用 -f 选项跳过确认。

程序清单 3-2 说明，Vagrant 可以修改 VM 的 VMX 文件，进行我们需要的更改。我们使用 config.vm.provider 代码块实现。顺便提一句，memsize 属性的单位是 MB。注意，我们将创建一个名为 v（包含在两条竖线之间）的对象，专为这个 VM 更改设置。这个对象名只有 config.vm.provider 语句内的局部作用域，可以在定义其他 VM 时再次使用，在后面的例子中可以看到。执行 vagrant up 之后，将创建具备所需属性的 VM。在本书写作时，虚拟磁盘的大小和数量还无法控制，但是你的 Vagrant VM 将启动 40GB 的精简配置存储。

程序清单 3-2 更改默认的 Vagrantfile

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

# Vagrantfile API/syntax version. Don't touch unless you know what you're
doing!
VAGRANTFILE_API_VERSION = "2"
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "hashicorp/precise64"
  config.vm.provider :vmware_fusion do |v|
    v.vmx["memsize"] = 1024
    v.vmx["numvcpus"] = 2
  end
end
```

可以修改 VM 的资源是很好的。那么，更复杂的设置（如多个 VM）该怎么进行？Vagrant 也支持这种拓扑。当然，一定要有足够的 CPU 核心和 RAM 以支持想要使用的拓扑！例如，多 VM 设置对测试具有独立数据库服务器和前端服务器的实际部署很有用。程序清单 3-3 展示了多 VM Vagrantfile 设置的一个例子。

程序清单 3-3 多机器 Vagrantfile

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

# Vagrantfile API/syntax version. Don't touch unless you know what you're
doing!
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.define :first do |vm1|
    vm1.vm.box = "hashicorp/precise64"
    vm1.vm.hostname = "devops"
    vm1.vm.provider :vmware_fusion do |v|
      v.vmx["memsize"] = 1024
      v.vmx["numvcpus"] = 2
    end
  end

  config.vm.define :second do |vm2|
    vm2.vm.box = "hashicorp/precise64"
    vm2.vm.hostname = "vmware"
    vm2.vm.provider :vmware_fusion do |v|
      v.vmx["memsize"] = 1024
      v.vmx["numvcpus"] = 2
```

```
    end
  end
end
```

这一部署利用了多个 config.vm.define 代码块：对我们要创建的每个 VM 各使用一个。:first 和 :second 是 Vagrant 在运行 vagrant status 等命令时用于标识 2 个 VM 的标签。这些标签也用于通过安全外壳（SSH）连接到 VM 时——例如，vagrant ssh first。如果你熟悉 Ruby，就会注意到这些标签是 Ruby 符号。包围在管道符号中的名称（例如，|vm1|）表示提供 vagrant 用于构建和定义 VM 信息的对象。对象名可以和符号相同（例如，first.vm.box…），但是不一定是这样。

当你想要部署超过 2 个 VM 时，使用这种语法可能有些乏味。幸好，因为 Vagrant 是用 Ruby 编写的，可以使用该语言的特性（如列表、循环和变量）优化 Vagrantfile 代码。程序清单 3-4 展示了我从 Cody Bunch 和 Kevin Jackson 的《OpenStack Cloud Computing》中学到的一些优化技巧。

程序清单 3-4　优化的多机器 Vagrantfile

```
# -*- mode: ruby -*-
# vi: set ft=ruby :
servers = ['first','second']

Vagrant.configure("2") do |config|
  config.vm.box = "hashicorp/precise64"
  servers.each do |hostname|
    config.vm.define "#{hostname}" do |box|
      box.vm.hostname = "#{hostname}.devops.vmware.com"
      box.vm.provider :vmware_fusion do |v|
        v.vmx["memsize"] = 1024
        v.vmx["numvcpus"] = 2
      end
    end
  end
end
```

在文件的开头，我创建了一个名为 servers 的 Ruby 列表，它的元素是想要创建的 VM 名称。然后，使用 Ruby 列表迭代子 each 循环执行服务器列表中每个元素的 VM 定义。如果想要增加部署的 VM 数量，只要在列表中添加更多的条目即可。不是每个 VM 都必须使用相同的资源集，可以在 box.vm.provider 代码块中使用 if 语句选择：

```
      if hostname == "first"
        v.vmx["memsize"] = 3128
        v.vmx["numvcpus"] = 4
```

```
  elsif hostname == "second"
    v.vmx["memsize"] = 1024
  end
```

Vagrant 还有许多特性无法在本书中介绍，但是用这几个简单的命令就能构建我们在本书中使用的测试环境。如果想要更多地了解 Vagrant，一定要访问 Vagrant 的网站（http://www.vagrantup.com），阅读 Mitchell 的《Vagrant: Up and Running》一书。

3.3 用 Packer 创建映像

Packer 是 HashiCorp 的另一种帮助你为多平台开发自定义盒子的产品。假定你想要从同一个基本盒子出发，开发用于 Workstation/Fusion 和 ESXi 的 VM 映像，Packer 就可以实现。

Packer 使用 JavaScript 对象标记法（JSON）文件格式指定 Vagrant 盒子的配置（磁盘大小、内存等），一旦指定了相关的自动化参数（例如，Ubuntu preseed 文件），它将帮助你进行初始 OS 部署。

Packer 不仅对创建 Vagrant 盒子有用；它的主要用途是制作与流行云提供商格式（OpenStack、AWS 等）兼容的映像文件。但是，Packer 包含构造器功能，可以自动输出与 VMware Fusion/Workstation 和 VirtualBox 兼容的 Vagrant 盒子，可以使用 Puppet 和 Chef 等流行配置管理技术，自定义生成的映像。

我们不深入讨论 Packer，但是如果想要自己试验自定义 Vagrant 盒子的构建，我们希望你了解它。可以在 http://www.packer.io 上找到更多关于 Packer 的信息。如果想要查看可用于开发你自己的 VM 的 Packer 定义文件，Chef 团队在自己的 Github 账户上维护着一个名为 bento 的存储库：https://github.com/chef/bento。

3.4 管理源代码

源代码管理（SCM,Source Code Management）是 DevOps 环境中必不可少的元素。想象一下：如果你要把基础设施转换为代码，重要的是有一种回顾任何更改、在新更改引入问题（例如，在最好的情况下是定期出现不稳定的情况，在最糟糕的情况下引起停机）时回到文件不同版本的手段。有些人可能认为“容易”的方法是建立文件的多个拷贝，每个都使用唯一的名称（Vagrantfile1、Vagrantfile2、Vagrantfile01012015 等），但是接着当你想要使用这些文件并且试图记住所有文件的不同之处时，就会陷入文件重命名的麻烦之中。

开发组织中的不同团队很有可能已经使用某种 SCM 系统管理他们的工作（例如，软件开发人员保存其源代码，QA 团队管理测试脚本）。在你开始使用 SCM 技术时，和其他小组讨论最佳实践是值得的。

SCM 解决方案包括 SVN、Mercurial 等。Git 是 DevOps 社区中较为流行的 SCM 系统之一，所以，在本书中我们使用 Git。

使用 Git

Git 是一个分布式版本控制系统，这意味着建立中央存储库的一个本地拷贝，而不只是访问单独的文件。本地提交可以同步到中央服务器，保持环境中的一致性，用户总是可以从中央存储库获取最新的源代码版本。这种架构与传统的源代码管理系统不同，原来的系统中只有中央服务器有完整的存储库拷贝。

在实验本书中的例子时，建议使用免费的在线 Git 存储库，如 BitBucket、GitHub 和 Gitorious，作为存储代码的中心位置。每个站点都有自己的独特功能。例如，BitBucket 允许无限制的免费私有存储库。GitHub 是本书作者用于自有代码的在线存储库系统。但是，完全可以使用任何符合你的需求的系统，因为获取代码（克隆 / 拉取）和存储代码（推送）的方法在任何 Git 系统上都通用。

注意　对于生产环境中的项目，考虑使用公共存储库站点之前请咨询法律部门。虽然许多站点都提供私有存储库功能，但是公司的领导可能更愿意使用内部中央 Git 服务器。

创建第一个 Git 存储库

首先，用你喜欢的包管理器（例如，Mac OS X 上的 homebrew 或 macports，Ubuntu/Debian 上的 apt-get，Red Hat/CentOS/Fedora 上的 yum）安装 Git。对于 Windows 用户，GitHub 和 BitBucket 等流行在线存储库提供软件客户端，简化了与在线存储库系统的交互。另外，http://git-scm.com 网站维护用于 Windows 的一个 Git 二进制独立安装版本。

如果你使用 Linux 或者 Mac OS X，可以打开一个终端窗口试验下面的例子。Windows 用户必须使用随 Git 客户端安装的特殊 shell。示例的语法基于 Liunx/Mac，但是在 Windows 平台上的命令应该等价。

在开始编写代码之前，必须设置两个全局变量，让 Git 知道我们是谁。这不像存储库的本地拷贝那么重要，但是在向远程服务器推送代码时非常重要。这两个全局变量是你的电子邮件地址和用户名：

```
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
```

事实上，如果尝试使用 Git，在第一次提交时没有设置这些变量，Git 将提示你设置它们之后才能继续。

如果你试验过前面的 Vagrant 示例，就已经有了一个用于处理内容的目录。否则，创建一个新目录并在其中创建一个文本文件。从现在起，要在命令行提示符上确认处于该目录。

首先，初始化该目录以建立一个 Git 存储库：

```
git init
```

如果用显示隐藏文件的选项列出目录（Linux/Mac 上的 ls-a，或者 Windows 上的 dir/

A:H)，就会看到一个隐藏目录 .git。这个目录包含存储库的文件和特定设置。这些本地设置和我们前面设置的全局设置组合，使用如下命令可以确认：

```
git config -l
```

如果想要查看目录中文件的状态（文件是否已经加入存储库？最后一个命令之后有无更改？等等），可以输入 git status，会看到类似程序清单 3-5 中展示的输出。

程序清单 3-5 Git 存储库状态

```
git-test $ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .vagrant/
        Vagrantfile

nothing added to commit but untracked files present (use "git add" to track)
```

最后一行最重要，它告诉我们需要跟踪存储库的文件以进行管理。注意 SCM 工具不会自动跟踪放入目录中的文件是很重要的。这一特性可以避免我们跟踪存储库中的垃圾文件，浪费空间。

我们告诉 Git 跟踪 Vagrantfile：

```
git add Vagrantfile
```

但是 .vagrant/ 目录没有必要跟踪，因为它只包含 Vagrant 用于设置 VM 的临时文件。可以创建一个 .gitignore 文件，明确地告诉 Git 忽略这个目录。使用你所喜欢的文本编辑器，创建有一个条目的 .gitigonore 文件：

```
.vagrant/
```

也可以使用简单的 echo 命令实现相同的功能。（Windows 用户需要使用包含在 Git 安装中特殊的 Git Shell 二进制文件才能正常工作。）

```
echo '.vagrant/' > .gitignore
```

如果再次运行 git status 命令，将会看到 Git 显示关于 .gitignore 文件的信息。发生了什么情况？记住，必须告诉 Git 如何处理存储库所能看到的任何文件或者目录，包括 .gitignore 文件。有两种方法可以处理 .gitignore 文件：

- 将 .gitignore 文件本身加入需要忽略的文件及目录列表。

■ 告诉 Git 跟踪 .gitignore 文件。

我将使用第二种选项，以便让使用存储库的其他人都能忽略相应的文件：

```
git add .gitignore
```

现在，如果再次检查存储库状态，会看到类似程序清单 3-6 中展示的输出。

程序清单 3-6　更新后的存储库状态

```
git-test $ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   .gitignore
        new file:   Vagrantfile
```

目录中的所有文件都已经准备好提交或者添加到非必要文件及目录的 .gitignore 列表。剩下的工作就是提交存储库更改：

```
git commit
```

这将自动打开一个文件编辑器，可以输入关于提交文件的细节。（默认使用 vi。）参见程序清单 3-7。

程序清单 3-7　你的提交消息

```
git-test $ git commit
  1 This is my first commit.
  2 # Please enter the commit message for your changes. Lines starting
  3 # with '#' will be ignored, and an empty message aborts the commit.
  4 # On branch master
  5 #
  6 # Initial commit
  7 #
  8 # Changes to be committed:
  9 #       new file:   .gitignore
 10 #       new file:   Vagrantfile
 11 #
```

必须输入一条消息；否则，提交将被撤销。如果不熟悉 vi，按下 I，输入一些文本，按下 Esc 键，然后输入 :wq 并按 Enter 键。如果不想使用文本编辑器，可以使用 git commit 命令的简短形式，加上 -m 选项，在同一行输入提交消息：

```
git commit -m "This is my first commit."
```

如果提交成功，应该看到如下输出：

```
[master (root-commit) d962cd6] This is my first commit.
 2 files changed, 119 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 Vagrantfile
```

使用中央 Git 服务器（远程服务器）

如果在 GitHub、BitBucker 或者任何公共 Git 存储库站点上开立了账户，就必须向站点提供计算机的 SSH 公钥，以便验证你的身份。每个站点完成这一操作的方法不同，查询文档找到对应的步骤。对于 Windows 用户，通常可以安装站点的客户端软件自动处理。Mac 和 Linux 用户必须使用 ssh -keygen 命令生成 SSH 公钥。

在远程服务器上正确配置 SSH 公钥之后，就可以在远程站点创建存储文件的存储库：这一过程称作推送（pushing）。在网站上创建远程存储库时，应该跳过 README 文件的自动生成。在存储库创建之后，站点将提供一个链接，可以使用它告诉本地存储库远程服务器的位置，该链接的标签往往标记为 origin。

在我的设置中，为存储库取了和本地存储库相同的名称。这是一个可选项，名称可以不同。可以使用 git remote 命令和 GitHub 提供的链接更新本地存储库设置：

```
git remote add origin git@github.com:DevOpsForVMwareAdministrators/git
```

使用 git config -l 命令，可以看到关于远程服务器位置的新数据：

```
remote.origin.url=git@github.com:DevOpsForVMwareAdministrators/git-test.git
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
```

现在，可以从本地存储库向远程存储库推送文件：

```
git push origin master
```

会看到类似程序清单 3-8 展示的输出。

程序清单 3-8　Git 远程推送结果

```
git-test $ git push origin master
Warning: Permanently added the RSA host key for IP address '196.30.252.129'
to the list of known hosts.
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 2.13 KiB | 0 bytes/s, done.
Total 4 (delta 0), reused 0 (delta 0)
To git@github.com:DevOpsForVMwareAdministrators/git-test.git
 * [new branch]      master -> master
```

图 3-1 展示了向远程服务器推送第一次提交的文件后 GitHub 上的存储库。

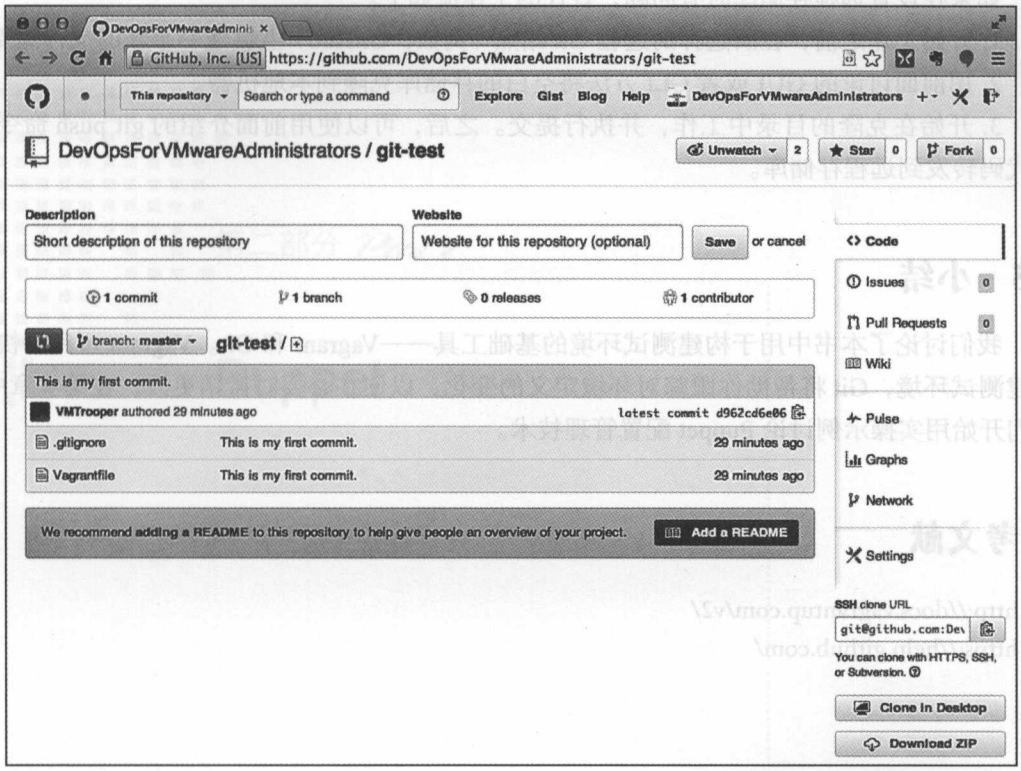


图 3-1　远程 Git 存储库

如果有了想要使用的远程存储库（正如我们为本书提供的样本），可以使用基于 CLI 或者 GUI 的方法将存储库克隆到本地机器。远程存储库系统应该有类似于图 3-1 左下角的选项。

根据使用的站点，基于 GUI 的方法有所不同。然而，在 GitHub 上，如果使用 Mac 或者 Windows 客户端，可以使用 Clone in Desktop 按钮，或者使用 Download Zip 按钮——这是一个包含存储库所有源代码的简单 zip 文件。如果使用 Windows 或 Mac 平台，建议使用 Clone in Desktop 选项，因为它将自动创建指向远程存储库的 Git 链接。

基于 CLI 的方法包括取得 SSH 或者 HTTP 克隆 URL 和使用 git clone 命令，如：

```
git clone https://github.com/DevOpsForVMwareAdministrators/git-test.git
```

执行上述命令之后，Git 将以存储库名称（在本例中是 git-test）创建一个目录，并将存储库的内容拷贝到这个目录中。你可以开始使用和更新代码，Git 远程链接将自动创建，正如前面提到的 Clone in Desktop 按钮那样。如果你有更改远程存储库的权限，可以执行 Git 推送，将任何本地更改转发到远程存储库。请求更改其他人的存储库不在本书讨论的范围内，但是，如果对此感兴趣，可以研究存储库的分支和拉取请求。这些功能的实现在不同的公共

存储库站点上各不相同，所以，必须查看所使用站点的对应文档。

如果在设置远程存储库时有问题，替代的工作流如下。

1. 开始工作之前，在所选择的远程 Git 站点（例如，GitHub）上创建一个新的空存储库。

2. 用前面讨论的 GUI 或者 CLI 方法将空白的存储库克隆到本地机器。

3. 开始在克隆的目录中工作，并执行提交。之后，可以使用前面介绍的 git push 命令将源代码转发到远程存储库。

3.5 小结

我们讨论了本书中用于构建测试环境的基础工具——Vagrant 和 Git。Vagrant 可用于快速构建测试环境，Git 将帮助你跟踪对环境定义的变化，以便在必要时撤销更改。在第 4 章中，我们开始用实操示例讨论 Puppet 配置管理技术。

参考文献

[1] http://docs.vagrantup.com/v2/

[2] https://help.github.com/

第二部分 Part 2

Puppet

- 第 4 章　Puppet 简介
- 第 5 章　Puppet 系统管理任务
- 第 6 章　用 Puppet 进行 VMware vSphere 管理

Chapter 4 第 4 章

Puppet 简介

Puppet 是由 Puppet Labs 开发的配置管理解决方案，有两种发行版本：开放源码版本和具有商业化支持的 Puppet Enterprise（企业版）。因为 VMware 配给功能要求 Puppet 企业版，所以我们将焦点放在这个特定的分发版本上。但是，本章讨论的大部分功能也适用于开放源码发行版本。

本章包含如下主题：

- Puppet 架构
- Puppet 资源
- Puppet 清单
- Puppet 模块

4.1 Puppet 架构

Puppet 可以管理的所有项目被称作 Puppet 资源。服务器软件包、配置文件和服务都是 Puppet 所能管理的资源。资源指令组合为 Puppet 清单（manifest）文件。如果想要部署 Apache Web 服务器，正如本章后面所讨论的，应该将 Apache Web 服务器软件包、配置文件和服务组合到同一个清单文件中。如果想将清单和配置文件、Web 内容文件等补充文件组合为容易分发的包，Puppet 提供了用于该用途的 Puppet 模块结构，模块可以包含多个清单及其他文件。我们将在本章后面深入讨论资源、清单和模块。

现在，我们关注用于你的环境的流行 Puppet 部署选项。Puppet 有两种部署拓扑：独立部署和主机 – 代理部署。

4.1.1　独立部署

在独立配置中，每台主机单独使用本地安装的 Puppet 代理软件管理。清单文件保存在本地，代理将配置管理指令编译为一个指令编目（catalog），供代理执行，参见图 4-1。

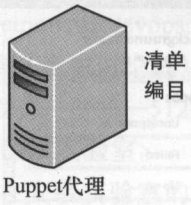


图 4-1　Puppet 独立拓扑

默认情况下，如果 Puppet 代理运行于守护进程模式，它每 30 分钟编译和执行编目。如果代理不以守护进程模式运行，可以用其他一些排程机制（如 cron）执行，或者使用 MCollective 等编排系统。系统管理员可以使用某种同步技术（如 rsync）保持清单更新，分发来自中央存储库的文件。

4.1.2　主机 – 代理部署

主机 – 代理拓扑由一台或者多台指定为主服务器的 Puppet 代理服务器组成。这些 Puppet 主机运行附加的二进制代码，集中管理所有代理节点，如图 4-2 所示。

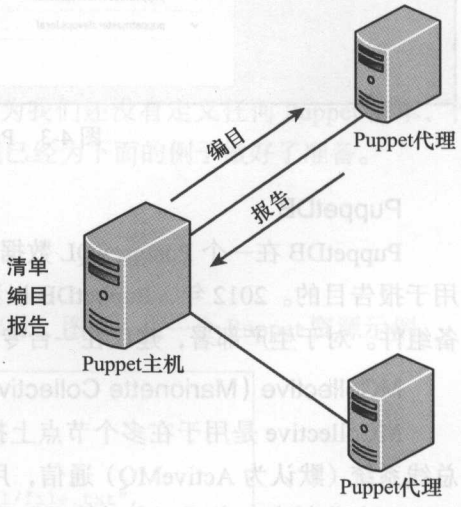


图 4-2　Puppet 主机 – 代理拓扑

这个拓扑中的一个显著差异是主机集中存储所有清单，通过本地存储的 site.pp 文件（企业版是 /etc/puppetlabs/puppet/environments/production/manifests，开源版是 /etc/puppet/manifests）中找到的条目确定每个代理执行的 Puppet 代码。主机将清单编译为编目并将其传输给代理执行。然后，代理向主机报告 Puppet 运行状态。

Puppet 主机组件

除了 Puppet 主机二进制代码之外，主服务器还将运行如下组件：

- 控制面板
- PuppetDB
- MCollective 编排
- 云配给

控制面板

控制面板是汇聚 Puppet 代理活动报告的 Web 界面。从这里，系统管理员可以执行创建 / 分配 / 删除组和类、临时 Puppet 代理执行等任务。控制面板组件只存在于 Puppet 企业版中，图 4-3 展示了 Puppet 企业版控制面板。

系统管理员可以在控制面板首页看到 Puppet 主机目前管理的节点列表。用户可以用页面左上角的菜单项执行前面列出的各种管理任务。

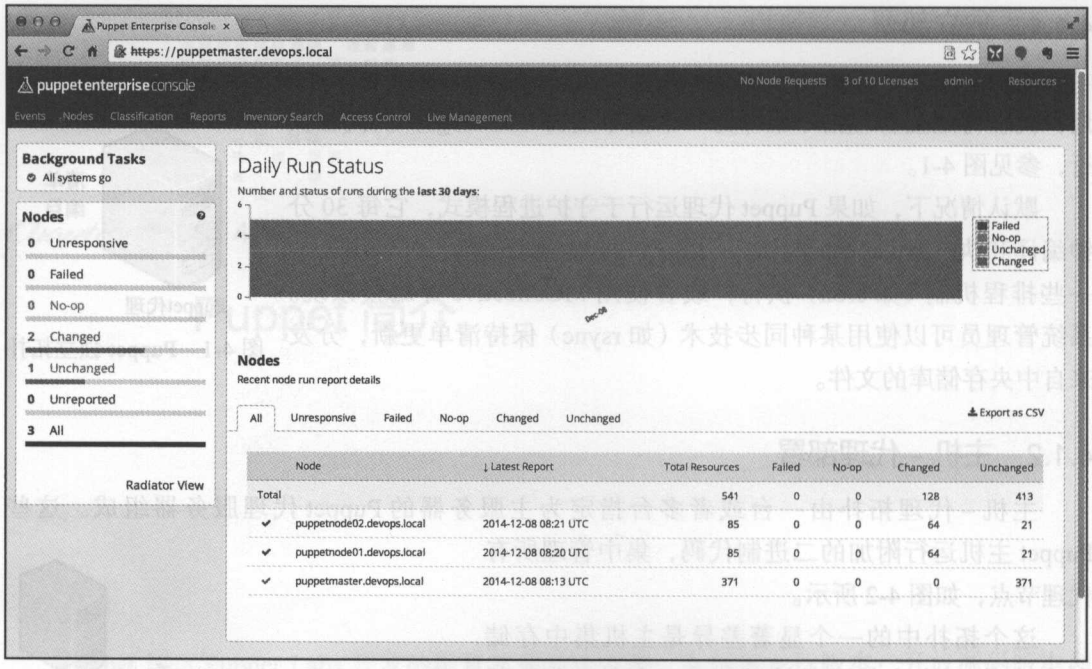


图 4-3 Puppet 企业版控制面板首页

PuppetDB

PuppetDB 在一个 PostgreSQL 数据库中以索引的持久化格式保存编目、fact 变量和报告，用于报告目的。2012 年，PuppetDB 以独立模块的方式推出，现已成为 Puppet 主机架构的必备组件。对于生产部署，建议在一台专用服务器上安装 PuppetDB 组件。

MCollective（Marionette Collective）

MCollective 是用于在多个节点上执行任务的编排软件。MCollective 组件通过一个消息总线系统（默认为 ActiveMQ）通信，用于环境中各个节点的异步管理。MCollective 包含在 Puppet 企业版中，实现“实时管理”功能，如触发临时 Puppet 运行。但是，MCollective 也可以用于开源版 Puppet。

云配给

这一功能可以将云工作负载配给到 VMware vSphere、Amazon EC2 和 Google Compute Engine 环境。管理任务包括创建新虚拟机（VM）、在 VM 上自动部署 Puppet 代理和将 VM 添加到类，使其可以自动执行 Puppet 代码。VMware vSphere 配给只能用 Puppet 企业版实现。

4.2 准备 Puppet 测试实验室

第 4 ~ 6 章中的例子以主机 – 代理拓扑使用 Puppet 企业版。该环境应该包含 3 个虚拟机。

- Puppet 主机：Ubuntu（12.04 或更高）
- Puppet 代理 1：Ubuntu（12.04 或更高）
- Puppet 代理 2：CentOS（6.5 或更高）

我的虚拟机分别使用完全限定域名 puppetmaster.devops.local、puppetnode01.devops.local 和 puppetnode02.devops.local。设置环境的最快方法是使用 Vagrant（在第 3 章中讨论过）。一定要为 Puppet 主机 VM 提供至少 4GB RAM。Puppet 代理 RAM 分配可以少一些——512MB。本书的 GitHub 页面有一个样板 Vagrantfile，可以它为基础开始构建你的测试实验室。

从 Puppet Labs 网站下载 Puppet 企业版软件，按照说明部署软件。在安装实验室的主服务器时一定要选择 Puppet Master、Console、PuppetDB 和 Cloud Provisioning 选项。Puppet 代理节点只需要在安装时选择 Puppet Agent 选项，但是它们需要有指向 Puppet 主机的 DNS 或者 /etc/hosts 条目。默认情况下，代理节点将是用主机名 / 别名 puppet 搜索主机。本书的 GitHub 页面上还包含了可用于自动化 Puppet 企业版安装的应答文件样板。

安装完成时，Puppet 代理节点将等待 Puppet 主机签署证书请求并开始通信。也可以登录到主服务器执行如下命令：

```
puppet cert sign --all
```

上述命令将自动启动代理商的 Puppet 运行。因为我们还没有定义任何 Puppet 清单，代理只会向主机报告它们的 fact 变量。现在，你的环境已经为下面的例子做好了准备。

4.3　Puppet 资源

资源是 Puppet 配置管理脚本（即清单）的组成部分。图 4-4 是一个 Puppet 资源示例。

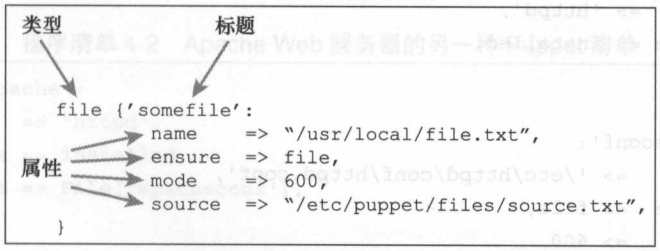


图 4-4　Puppet 类型文件资源

资源由以下一些元素组成。

- 类型：将要配置的资源类型（软件包、文件等）。有许多种资源类型，可以在 Puppet Labs 网站上看到更完整的类型列表（http://docs.puppetlabs.com/learning/ral.html）。
- 标题：配置管理脚本引用资源的方式。
- 属性：为资源指定的设置。不同资源类型有不同的属性。
- 提供者：与配置管理脚本引用资源方式相关的工具。

提供者通常不在资源陈述中明确声明。Puppet 根据资源类型自动使用正确的提供者（例如，在基于 Red Hat 的操作系统上安装软件包时使用 yum）。如果定义了自己的 Puppet 类型，也必须指定使用的提供者。开发自定义类型超出了本书讨论的范围，但是知道 Puppet Labs 有这类灵活性是很好的事。

在进一步学习之前，我们还要了解一个特殊的资源属性：namevar。这是 Puppet 用于标识所声明资源的值，必须是唯一的。如果声明了 name 属性，则 namevar 取该属性的值。如果没有声明 name 变量，namevar 被设置为资源的标题。为什么这一点很重要？ Puppet 必须有唯一的 namevar 值。否则，在执行 Puppet 清单时将会出错。明确地命名资源是一个很好的主意。

4.4 Puppet 清单

清单是资源的集合，它们是管理员用于管理环境中服务器的配置脚本。系统管理员可以定义特殊属性，将不同的资源关联在一起，这为管理员提供了构造清单的灵活性。

Puppet 不一定按照编写的顺序处理清单。所以，如果资源应该按照某种优先顺序处理，Puppet 有一些称作元参数（metaparameter）的特殊属性，可以定义清单中资源之间的关系。这些属性是 before、require、notify 和 subscribe。元参数的使用是 Puppet 特有的，重要的是在清单中有优先执行顺序时记得它们。程序清单 4-1 展示了 require 和 subscribe 语句的用法示例。

程序清单 4-1 Apache Web 服务器的 Puppet 清单

```
package { 'apache':
        name   => 'httpd',
        ensure => installed,
}

file {'apacheconf':
        name    => '/etc/httpd/conf/httpd.conf',
        ensure  => file,
        mode    => 600,
        source  => '/etc/puppet/files/httpd.conf',
        require => Package['apache'],
}
service {'apache':
        name      => 'httpd',
        ensure    => running,
        enable    => true,
        subscribe => File['apacheconf'],
}
```

程序清单 4-1 中的清单将在 CentOS 服务器（也就是 Puppet 代理 2）上工作。你需要将代码输入代理服务器上的清单文件，一定要使用文件扩展名 pp。该清单需要一个与你所运行的 CentOS 版本兼容的 Apache 配置文件（参见文件资源）。一定要更改资源的来源属性，反映 Apache 配置文件的存储位置。即使配置文件和清单在同一个目录，也要提供明确的路径。

我们在这里暂停一下，让你有机会在测试机上执行代码。

记住，执行清单的命令如下：

```
puppet apply filename.pp
```

如果想要观察 Puppet 代码做了什么，可以使用 apply 命令的 -d 选项。

现在，回到样板代码：文件来源中的 require 元参数确保指定资源（Package["apache"]）在自身资源（file('apacheconf')）部署之前存在。这样做是有道理的，因为试图为不存在的服务部署配置文件毫无意义！

subscribe 元参数的表现和 require 类似，但是有一个额外的功能：Puppet 将监控目标资源的任何更改，并执行相应的操作作为响应。对于服务资源，这意味着在检测到配置文件中的更改后重新启动服务。

注意　你可能已经注意到，我在表明现有资源顺序关系的 Puppet 类型名称上使用大写（例如，subscribe => File['apacheconf']），这是 Puppet 的惯例。在第一次声明资源时，类型名是小写的，以后在清单中引用时类型名的第一个字母为大写。

before 和 notify 元参数分别与 require 和 subscribe 相反。如果在清单中使用 before 和 notify，则如程序清单 4-2 所示。

程序清单 4-2　Apache Web 服务器的另一种 Puppet 清单

```
package { 'apache':
        name   => 'httpd',
        ensure => installed,
        before => File['apacheconf'],
}

file {'apacheconf':
        name    => '/etc/httpd/conf/httpd.conf',
        ensure  => file,
        mode    => 600,
        source  => '/etc/puppet/files/httpd.conf',
        notify => Service['apache'],
}

service {'apache':
```

```
        name      => 'httpd',
        ensure    => running,
        enable    => true,
    }
```

还有另一种定义关系的方法——使用链接箭头（→表示 before/require 关系，~> 表示 notify\subscribe 关系）。这些箭头可以放在资源声明之间，代替元参数语句，如程序清单 4-3 所示。

程序清单 4-3 使用链接箭头编写的 Puppet 清单

```
package { 'apache':
        name   => 'httpd',
        ensure => installed,
}
->
file {'apacheconf':
        name    => '/etc/httpd/conf/httpd.conf',
        ensure  => file,
        mode    => 600,
        source  => '/etc/puppet/files/httpd.conf',
}
~>
service {'apache':
        name      => 'httpd',
        ensure    => running,
        enable    => true,
}
```

作为资源之间放置关系箭头的替代方法，可以直接声明资源，并在清单的最后加入资源标题以及标题之间的箭头，如：

```
Package['apache'] ->

File['apacheconf'] ~>

Service['apache']
```

定义资源顺序没有首选的方法。但是，建议你的团队确定一种最易于所有人理解的方法。

可以在清单中创造性地放置资源，但是我们建议在编写 Puppet 代码时采用合理的顺序。（例如，如果资源 B 依赖于资源 A，在清单中资源 A 应该在 B 之前列出。）这能够使其他团队成员更容易理解工作流的意图。而且，在一年之后，甚至连你也可能忘记编写清单的思想过程。所以，要尽可能保持简单。

注意 你可能觉得奇怪，“为什么不按照我们写的顺序执行 Puppet 代码？”
长期以来，Puppet 用户已经提出了类似的反馈，从 Puppet 3.3.0 开始，Puppet 已经实现了资源清单顺序分析（MOAR）。MOAR 最终会成为未来开源 Puppet 和 Puppet 企业版的默认行为。但是，喜欢当前行为（也称为标题散列）的用户也可以启用。

前面的例子展示了编写 Puppet 清单的精髓：软件包 – 文件 – 系统。指定了应该部署的软件包、该软件包相关的配置或者内容文件以及软件包的服务状态（运行、停止等）。在开始编写 Puppet 代码时，这是应该遵循的极好模式。

条件表达式和变量

前面展示的基本清单很强大。但是，你可能想要根据执行清单的系统的属性修改清单的执行方式。而且，如果软件包的名称更改，代码的编辑可能有些乏味，因为软件包名称的更改很有可能影响其他资源（配置文件路径、服务状态命令等）。

因为上述原因和其他一些原因，Puppet 允许使用变量和条件表达式。除此之外，安装 Puppet 时还包含一个名为 facter 的二进制模块，它自动生成描述服务器特性（是物理机器还是虚拟机器？ IP 地址是？操作系统是？等）的一组变量——这组变量被称为事实（fact）。事实可以从命令行运行 facter 命令访问。在运行 facter 命令时可以指定单独的 fact 变量（facter is_virtual、facter ipaddress、facter operatingsystem 等）。运行时，Puppet 清单可以访问所有 fact 变量。

在清单文件中，所有变量标签都以 $ 作为前缀，包括自动生成的 fact 变量（$is_virtual 等）。变量不需要声明 string 或者 int 等类型，就像 Ruby 一样，Puppet 通过初始化值理解你所声明的变量类型。

在代码中使用包含字符串值的变量时，必须使用双引号而不是单引号。将变量数据包含在字符串中称作“内插”（interpolation），如下例中 $webserver 变量的用法：

```
$webserver='httpd'
file {'apacheconf':
        name    => "/etc/$webserver/conf/$webserver.conf",
        ensure  => file,
        mode    => 600,
        source  => "/etc/puppet/files/$webserver.conf",
        require => Package['apache'],
}
```

如果担心在单引号 / 双引号或者其他 Puppet 语法项目上犯错，Geppeto 和 Sublime Text 的 Puppet 软件包等集成开发环境（IDE）支持 Puppet 语法高亮显示，由此可以提供帮助。这些工具有助于避免引号和其他 Puppet 惯例上的错误。

条件语句能够全面利用 Puppet 变量的能力。Puppet 中支持的条件语句有 if 和 case。回到之前的 Apache 部署示例，Apache 软件包根据你所部署的操作系统风格而有所不同。例如，在基于 Red Hat 的系统上，Apache 二进制文件被称作 httpd。在基于 Debian 的系统上，软件包被称作 apache2。我将使用 fact 变量 operatingsystem 确定清单中使用的对应设置。

下面是使用 if 条件的例子：

```
if $operatingsystem == 'centos' {
      $webserver= 'httpd'
      $confpath = "/etc/$webserver/conf/$webserver.conf"
  }
elsif $operatingsystem == 'ubuntu': {
      $webserver= 'apache2'
      $confpath = "/etc/$webserver/$webserver.conf"
  }
else {
  fail("Unsupported OS")
}
```

下面是使用 case 条件语句的例子：

```
case $operatingsystem {
  centos: {
      $webserver= 'httpd'
      $confpath = "/etc/$webserver/conf/$webserver.conf"
  }
  ubuntu: {
      $webserver= 'apache2'
      $confpath = "/etc/$webserver/$webserver.conf"
  }
  default: {
      fail("Unsupported OS")
  }
}
```

在前面的例子中，我使用 fact 变量 operatingsystem 确定声明的两个变量值。如果运行脚本的服务器使用的不是基于 Red Hat 或者 Debian 的操作系统，则生成一个清单执行错误的通知，因为那是一个意料之外的环境。

对于在一两个值之间做决定的条件检查，if 语句可能更合适。但是，如果需要从很长的取值列表中选择，就应该使用 case 语句代替。

添加 case 语句和变量之后，前面的 Apache 清单将类似于程序清单 4-4。

程序清单 4-4 使用条件语句支持多平台的 Puppet 清单

```
case $operatingsystem {
  centos: {
```

```
        $webserver= 'httpd'
        $confpath = "/etc/$webserver/conf/$webserver.conf"
    }
    ubuntu: {
        $webserver= 'apache2'
        $confpath = "/etc/$webserver/$webserver.conf"
    }
    default: {
        fail("Unsupported OS")
    }
}

package { 'apache':
        name   => $webserver,
        ensure => installed,
}

file {'apacheconf':
        name    => $confpath,
        ensure  => file,
        mode    => 600,
        source  => "/etc/puppet/files/$webserver.conf",
        require => Package['apache'],
}

service {'apache':
        name      => $webserver,
        ensure    => running,
        enable    => true,
        subscribe => File['apacheconf'],
}
```

4.5 Puppet 模块

Puppet 模块使系统管理员可以组合有用的 Puppet 代码，在未来的清单中重用。我们考虑 Apache 部署示例，每当部署新的 Apache 主机时，都需要在新清单中一次又一次地复制和粘贴同样的代码。这种方法的问题是，如果想要改变部署 Apache 的方法，就必须编辑多个 Puppet 清单才能完成所需的更改。

为了促进高效的代码重用，Puppet Labs 设计了 Puppet 类功能，其概念类似于面向对象编程语言中的类。但是不要担心，我们不会过于深入计算机科学理论。

类可以在清单中使用，但是打包成一个可以供其他清单使用的模块更有用。

我们先来看看如何建立一个 Puppet 类。首先修改 Apache 部署清单，如程序清单 4-5 所示。

程序清单 4-5 从现有清单创建一个 Puppet 类

```
class apache {
  case $operatingsystem {
    centos: {
        $webserver= 'httpd'
        $confpath = "/etc/$webserver/conf/$webserver.conf"
    }
    ubuntu: {
        $webserver= 'apache2'
        $confpath = "/etc/$webserver/$webserver.conf"
    }
    default: {
        fail("Unsupported OS")
    }

  }
  package { 'apache':
       name   => $webserver,
       ensure => installed,
  }

  file {'apacheconf':
       name    => $confpath,
       ensure  => file,
       mode    => 600,
       source  => "/etc/puppet/files/$webserver.conf",
       require => Package['apache'],
  }

  service {'apache':
       name      => $webserver,
       ensure    => running,
       enable    => true,
       subscribe => File['apacheconf'],
  }
}
include apache
```

真难！我们添加了整整 3 行。严肃地说，将清单更改成一个类所需要做的就是在清单之前加入“类名 {”，在最后加入“}”。然后，告诉清单包含该类。

如果想为类指定参数，也是可以的。例如，如果想要命令 Puppet 部署特定的登录页面，可以这样定义类：

```
class apache($landingpage = "/etc/puppet/files/index.html") {
```

```
<your manifest code>
}
include apache
```

我们定义了一个参数（$landingpage），并用赋值运算符设置了默认值。如果清单作者没有指定任何参数（就像前面所做的），则使用默认值。

同样，类是很好的概念，但是如果想要轻松地重用它们，必须将其转换为 Puppet 模块。模块必须放在 modulepath 设置指定的位置，用户可以配置这个设置。如果使用 Puppet 企业版，模块将放置在 /etc/puppetlabs/puppet/environments/production/modules、/etc/puppetlabs/puppet/modules 或者 /opt/puppet/share/puppet/modules 中。如果使用开源版 Puppet，模块将放置在 /etc/puppet/modules 或 /usr/share/puppet/modules 中。可以在主服务器上用 puppet config print modulepath 命令验证系统的模块路径设置。

4.5.1　Puppet Forge

在深入模块创建之前，我要介绍 Puppet Lab 的在线共享模块存储库：Puppet Forge（可以从 http://forge.puppetlabs.com 访问）。你几乎可以将其视为 Puppet 模块的 GitHub。在 Forge 网站上，可以找到 Puppet 用户社区和 Puppet Labs 共享及支持的模块。

在 Forge 网站上开立一个免费账户以便保留用户名。建议使用和 GitHub 账户相同的用户名，我们将在模块创建时使用 Puppet Forge 用户名，所以要保持它的方便性。

现在，你有了 Puppet Forge 账户，可以在 GitHub 上创建一个存储库。由于我们要创建的是一个 Web 服务器模块，可以将存储库命名为“apache”。在以后设置为本地存储库的远程映像之前，这个存储库只是一个占位符。

4.5.2　创建第一个 Puppet 模块

模块需要一种特殊的树结构，Puppet 清单才能正确访问它们。好消息是，不需要人工创建目录结构，Puppet 包含一个正确生成模块目录文件夹的命令。

首先，转到为本书示例选择的目录。然后，使用如下命令开始模块目录路径创建：

```
puppet module generate username-modulename
```

对于没有输入的值，接受默认值或者留空都没有问题。当该工具要求输入源代码存储库时，可以输入前面创建的 GitHub 存储库的 HTTP 链接。如果在任何输入中出错，可以编辑生成的 metadata.json 文件。

继续之前，我们需要对生成的目录结构进行少量更改，以便在例子中正确使用。puppet module generate 命令假定你构建一个将在 Puppet Forge 网站上共享的模块。目录的命名惯例采用用户名 - 模块格式，从而使代码无法在本地正常工作。而且，我们漏掉了两个重要的子目录：files 和 templates，现在修复这些问题。

首先，将目录名从“用户名 - 模块名”改为仅使用模块名。例如，将目录名 vmtrooper-

apache 改为 apache。接下来，在 apache 目录下创建 templates 和 files 子目录。图 4-5 展示了正确目录结构的一个例子。如果不做这些更正，在本章的示例代码上就会遇到困难。

manifests 是重要文件夹，包含与模块相关的所有 Puppet 代码。标准模块的惯例是有一个定义类的特殊清单文件 init.pp。你可以在其他清单文件中放入代码。但是，清单目录中至少应该存在 init.pp 文件。我们将展示一个示例，但是首先描述组成模块的其他重要组件。

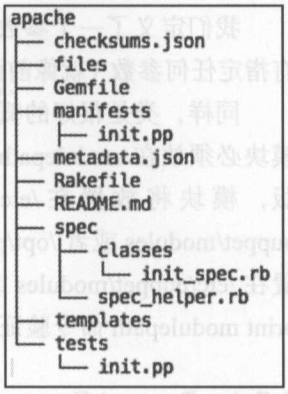


图 4-5 Puppet 模块目录结构

- files：在清单执行期间分发的所有文件（例如，apache 配置文件）必须放置在这个目录下。正如在 4.5.3 节的 init.pp 示例中所见，这使我们可以使用新的文件访问方法，该方法更清晰，有助于避免路径问题。
- lib：如果编写了任何自定义 fact 变量或者类型，将代码放在这个目录下。
- spec：这个文件夹包含用于 lib 文件夹中自定义代码的测试。
- templates：如果想要部署根据运行时信息更改的软件包配置或者内容文件，将它们和一些嵌入 RuBy（ERB）代码一起放在这个目录。
- tests：这个文件夹包含你想要用来说明模块使用方法的示例。
- metadata.json：这个文件包含 semver 格式的模块版本名称，以及 puppet module generate 命令提供的其余数据。

现在是为你的模块初始化 Git 存储库，指定 GitHub 存储库为模块的远程存储库，并执行第一次提交和来源回推的好时机。你的虚拟机（VM）可能还没有安装 Git，所以，使用操作系统的软件包管理器安装。在本章内容的推进中，一定要定期提交你的工作，以便回退任何造成问题的更改。如果完成这些任务需要复习相关的 Git 命令，可以回到第 3 章。

4.5.3 Puppet 模块初始化清单（init.pp）

如程序清单 4-6 所示，我们已经使用过的清单代码将经过小的修改之后进入 init.pp 文件。

程序清单 4-6 Apache Web 服务器模块的 init.pp 文件

```
class apache {
  case $operatingsystem {
    centos: {
        $webserver= 'httpd'
        $confpath = "/etc/$webserver/conf/$webserver.conf"
    }
    ubuntu: {
        $webserver= 'apache2'
        $confpath = "/etc/$webserver/$webserver.conf"
```

```
    }
    default: {
        fail("Unsupported OS")
    }

}

    package { 'apache':
        name   => $webserver,
        ensure => installed,
    }

    file {'apacheconf':
        name    => $confpath,
        ensure  => file,
        mode    => 600,
        source  => "puppet:///modules/apache/$webserver.conf",
        require => Package['apache'],
    }

    service {'apache':
        name      => $webserver,
        ensure    => running,
        enable    => true,
        subscribe => File['apacheconf'],
    }
}
```

整个类定义都将进入 init.pp 文件，在我的默认安装中，该文件将放置在 /etc/puppetlabs/puppet/modules/apache/manifests/ 目录。

继续之前，我们将焦点放在文件资源上，如果观察来源属性，你会看到和前面不同的路径，该路径采用了特殊的 puppet URL 格式。

使用模块的另一个好处是，puppet 将模块路径几乎当作文件服务器处理。前缀 puppet:///modules/apache/ 告诉 puppet 在 Puppet 主机 modulepath 中找到的 apache 模块子目录 files 中搜索源文件（也就是 /etc/puppetlabs/puppet/modules/apache/files）。如果在 files 目录中创建一个目录，只需将其插入文件名之前，如 puppet:///modules/apache/subdirectory/$webserver.conf。

所以，不管你的清单从哪个系统中运行，都没必要担心放置文件的完整路径。只需将其放在 /etc/puppetlabs/puppet/modules/apache/files/ 中即可。

4.5.4 模板

在结束模块的讨论之前，我们研究一下如何使用模板，使部署更加灵活。我们可以使用

单一配置文件模板代替用于 CentOS 和 Debian 的不同文件。但是，对各种选项和 Apache 配置的讨论超出了本书的范围。我们将在讨论中使用更简单的示例，如 index.html 文件的内容。默认网页将告诉用户 Web 服务器上运行的操作系统种类，在实践中，这些简单的示例说明了模板的用途。

首先，在我们的 apache 模板 templates 目录下创建 index.html.erb 文件，包含如下内容：

```
<html><body><h1>Puppet Rocks!</h1>
<p>This is the default web page for
<%# Print the correct Operating System name with the right article i.e. "a"
  vs "an"%>
<% if @operatingsystem[0].chr =~/[AEIOU]/ %>
  <%= "an" %>
<% else %>
  <%= "a" %>
<% end %>
<b><%= @operatingsystem %></b> server.</p>
<p>The web server software is running and content has been added by your
  <b>Puppet</b> code.</p>
</body></html>
```

注意，Ruby 代码块有不同的前缀：

- <%= 前缀用于输出到从模板生成的文件中的任何值（例如，操作系统名和冠词 a 或者 an）。
- <%# 用于向文件编辑者解释模板代码的作用。
- <% 用于条件语句或者循环等 Ruby 代码。如果你想要根据数组、列表、散列等结构中的条目输出多行值，这些语句很有用。我们包含在 index.html.erb 中的条件语句利用 Ruby 的正则表达式功能，验证操作系统名称以元音或者辅音字母开头。

接下来，我们修改 apache 类，添加一个用于默认网页的文件资源。我们将添加一个关系元参数，使 HTML 文件不在 Apache Web 服务器完全启动运行之前部署。而且，我们必须知道 index.html 文件的部署路径，因为 CentOS 和 Ubuntu 的默认路径不同。我们将添加一个新变量 htmlpath 以考虑这个情况，Puppet 代码现在如程序清单 4-7 所示。

程序清单 4-7 更新后的 Apache 类，包含 index.html ERB 模板

```
class apache {
  case $operatingsystem {
    centos: {
        $webserver= 'httpd'
        $confpath = "/etc/$webserver/conf/$webserver.conf"
        $htmlpath = "/var/www/html/index.html"
    }
    ubuntu: {
        $webserver= 'apache2'
```

```
        $confpath = "/etc/$webserver/$webserver.conf"
        $htmlpath = "/var/www/index.html"
    }
    default: {
        fail("Unsupported OS")
    }

  }

  package { 'apache':
      name   => $webserver,
      ensure => installed,
  }

  file {'apacheconf':
      name    => $confpath,
      ensure  => file,
      mode    => 600,
      source  => "puppet:///modules/apache/$webserver.conf",
      require => Package['apache'],
  }

  service {'apache':
      name       => $webserver,
      ensure     => running,
      enable     => true,
      subscribe => File['apacheconf'],
  }
  file {'apachecontent':
      name    => $htmlpath,
      ensure  => file,
      mode    => 644,
      content => template('apache/index.html.erb'),
      require => Service['apache'],
  }
}
```

使用静态文件和带有文件资源的模板有几处不同。在模板中，我们使用文件资源的content属性而不是source属性。content属性通常用于指定文件的真正内容。如果使用template关键字和content属性，Puppet将在处理ERB代码时在目标文件中插入模板文件的内容。

content属性不是唯一的区别；和template关键字一起指定的路径也不同于源属性中使用的路径，其格式如下：

<模块名>/<模块文件名>

在我们的样板代码中，模块的名称为 apache，模板文件名为 index.html.erb。模板文件必须放在模块路径的 templates 目录下（/etc/puppetlabs/puppet/modules/apache/templates/）。

4.5.5 使用 Puppet 模块

在原始清单中，部署 Apache Web 服务器所需的唯一代码如下：

```
include apache
```

如果定义另一个类作为模块的一部分（例如，vhost），它可以通过如下调用在清单中使用：

```
include apache::vhost
```

4.5.6 最后一步：版本控制提交

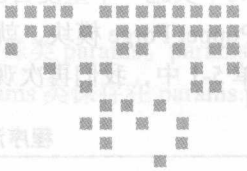
我们已经为 Apache Web 服务器模块做了很多工作。在结束本章之前，不要忘记提交更改，并将提交推送到远程存储库。我们将在第 5 章中从远程存储库克隆这一模块使用。

4.6 小结

本章介绍了相当多的 Puppet 特性和概念。我们首先讨论了 Puppet 架构，然后介绍了 Puppet 资源——Puppet 管理的对象（例如，软件包、配置文件和服务）。接着，我们讨论这些资源如何组合为指令文件——Puppet 清单。本章还解释了如何使用 Puppet 模块（相关清单的集合）根据用户规格部署应用程序。在第 5 章中，你将尝试 Linux-Apache-MySQL-PHP（LAMP）栈部署。但是，在继续学习之前，如果想要实践更多的 Puppet 技术基础知识，可以访问 Puppet Labs 网站的 Learning Puppet VM。

参考文献

[1] “Puppet 3 Reference Manual”: http://docs.puppetlabs.com/puppet/3/reference/



第 5 章 Chapter 5

Puppet 系统管理任务

Puppet 对于快速部署和配置关键应用很有用。本章，我们研究包含 Web 服务器、应用服务器和数据库服务器层次的多层部署。这种应用程序组合常被称作 LAMP（Linux-Apache-MySQL-PHP）部署。根据实际安装的模块，缩略语中的各个字母可能变化（例如，用 nginx[发音为“engine X”] 代替 Apache 时缩写为 LEMP）。

我们将在第 4 章中完成的 Apache Web 服务器部署的基础上充实整个 LAMP 栈。在本章的示例中，使用第 4 章中介绍的 3 节点设置：一个 Puppet 主服务器（puppetmaster.devops.local）和 2 个 Puppet 代理服务器（puppetnode01.devops.local 和 puppetnode02.devops.local）。模块代码必须保存在 Puppet 主服务器的 modulepath 下。如果想要复习如何获取正确路径，参考第 4 章。

本章包含如下主题：

- Web 层
- 用数据分离优化 Web 层
- 应用层
- 数据库层

5.1 用数据分离优化 Web 层

在讨论多层应用组件之前，我们先介绍编写更好 Puppet 模板的建议方法：数据分离。这一方法包括将清单中执行的指令与指令所使用的数据分离。这些数据将保存在一个中心位置，以便模块中的多个清单重用这些数值，而不是在多个文件中多次定义这些值。这将帮助

你在 Puppet 代码中实施“不重复自己的工作”（DRY）方法，减少数据更改时需要管理的源代码。如果检查 Puppet Forge 模块，就会注意到它们采用了这一方法。

在程序清单 5-1 中，我们再次观察第 4 章中的 Apache 类，研究如何实现数据分离。

程序清单 5-1 原始 Apache 模块

```
class apache {
  case $operatingsystem {
    centos: {
        $webserver= 'httpd'
        $confpath = "/etc/$webserver/conf/$webserver.conf"
    }
    ubuntu: {
        $webserver= 'apache2'
        $confpath = "/etc/$webserver/$webserver.conf"
    }
  }

  package { 'apache':
      name   => $webserver,
      ensure => installed,
  }

  file {'apacheconf':
      name    => $confpath,
      ensure  => file,
      mode    => 600,
      source  => "puppet:///modules/apache/$webserver.conf",
      require => Package['apache'],
  }

  service {'apache':
      name      => $webserver,
      ensure    => running,
      enable    => true,
      subscribe => File['apacheconf'],
  }
}
```

在当前状态下，我们的 Puppet 代码相当容易移植和维护。但是，实际的模块可能有多个类，每个类都有自己的清单文件，如果希望这些类根据操作系统决策，必须将条件逻辑从 init.pp 复制到其他每个清单文件。

在多个清单文件中维护数据选择逻辑可能令你和同事有些头痛。因此（还有其他很好的理由），建议将 Puppet 清单代码与其使用的数据分离。

5.1.1　参数类（params.pp）

模块作者可以利用的数据分离方法之一是创建一个特殊类 params。params 类（参见程序清单 5-2）包含条件逻辑和清单作为操作输入的数据。params 类保存在 params.pp 文件中，该文件保存在模块的 manifests 目录，与 init.pp 文件在一起。

程序清单 5-2　参数类

```
#/etc/puppetlabs/puppet/module/apache/manifests/params.pp
class apache::params {
  case $::operatingsystem {
    'CentOS': {
        $webserver= 'httpd'
        $confpath = "/etc/$webserver/conf/$webserver.conf"
        $htmlpath = '/var/www/html/index.html'
    }
    'Ubuntu': {
        $webserver= 'apache2'
        $confpath = "/etc/$webserver/$webserver.conf"
        $htmlpath = '/var/www/index.html'
    }
    default: {
        fail("The ${module_name} does not support this Operating System")
    }
  }
}
```

该类的名称以模块名作为前缀，指出数据的作用域。被其他文件引用时，params 文件中的所有变量将加上 apache::params 前缀。你将会注意到，我明确地定义操作系统 fact 变量（$::operatingsystem）的作用域。这将帮助其他修改代码的管理员理解，我们有意使用系统事实代替 params.pp 文件其他位置声明的同名变量。即使没有在类中声明另一个 $operatingsystem 变量，在类中使用 fact 变量时明确指出变量作用域也是一个好的习惯。有助于代码移植性的最后一个更改是更新 case 语句的默认操作，为用户提供更有意义的错误信息。那么，我们的 init.pp 清单如何使用 params.pp 文件中的数据？我们来看看更新后的文件（参见程序清单 5-3）。

程序清单 5-3　更新后的 Apache 模块

```
#/etc/puppetlabs/puppet/module/apache/manifests/init.pp
class apache (
  $webserver= $apache::params::webserver,
  $confpath = $apache::params::confpath,
  $htmlpath = $apache::params::htmlpath,
) inherits apache::params
```

```
{
  package { 'apache':
      name   => $webserver,
      ensure => installed,
  }

  file {'apacheconf':
      name    => $confpath,
      ensure  => file,
      mode    => 600,
      source  => "puppet:///modules/apache/$webserver.conf",
      require => Package['apache'],
  }
  service {'apache':
      name      => $webserver,
      ensure    => running,
      enable    => true,
      subscribe => File['apacheconf'],
  }
  file {'apachecontent':
      name    => $htmlpath,
      ensure  => file,
      mode    => 644,
      content => template('apache/index.html.erb'),
      require => Service['apache'],
  }
}
```

Puppet 继续借鉴面向对象编程原则，使用 inherits 关键字表示主清单（init.pp）和 params 类（class apache {…} inherits apache::params）之间的关系。我建议的可选更改之一是将类定义变成参数化类，默认值对应于 params.pp 中的数据。我说“可选”是因为声明不带参数的类就已经足够了。inherits 语句使我们可以调用 params 文件中的变量名而无须明确指定作用域。同样，为了便于代码维护，强烈建议明确定义作用域。除了这些更改，清单的其余部分保持不变。

在继续之前，我们抓住这次机会，尝试以典型的主机 – 代理风格应用我们的新版本 apache 模块。

这是一个两步或者三步的过程：

1. 从远程存储库克隆 Apache Web 服务器模块，更正 Puppet 主机上的路径。（用 puppet config print modulepath 确认。）

2. 编辑 Puppet 主服务器的 site.pp 文件，并指定应该运行模块的主机。

3.（可选）告诉 Puppet 代理节点应用我们定义的 Puppet 代码，也可以等待下一次自动化

Puppet 代理执行，但是那有什么意思？

第 1 步应该很容易理解。提交迄今为止的更改，并将其提交到你的远程存储库。在 Puppet 主服务器上，使用 Git 将远程存储库克隆到 modulepath。然后，应该在 Puppet 主服务器的 modules 目录上得到如图 5-1 所示的树结构。

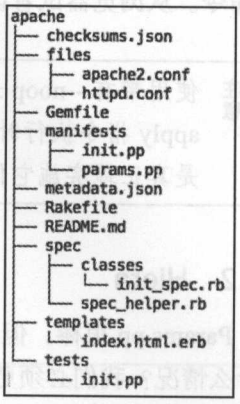


图 5-1　apache 模块的目录结构

第 2 步要求编辑 Puppet 主机上 /etc/puppetlabs/puppet/environments/production/manifests 目录中找到的 site.pp 文件。你会注意到，有一个默认条目，可以用作定义节点的模板。我们建立默认条目的两个拷贝，并按照两个 Puppet 节点命名：如果你遵循我的命名惯例，完全限定域名（FQDN）将是 puppetnode01.devops.local 和 puppetnode02.devops.local。确定哪个 Puppet 代理服务器是 Web 服务器并包含我们创建的 apache 模块。（在我的例子中，puppetnode01 是 Web 服务器）。site.pp 设置类似于程序清单 5-4。

程序清单 5-4　Puppet 代理服务器的 site.pp 条目

```
node default {
  # This is where you can declare classes for all nodes.
  # Example:
  #   class { 'my_class': }
}

node 'puppetnode01.devops.local' {
  class { 'apache': }
}

node 'puppetnode02.devops.local' {
   # We'll fill this in later
}
```

第 3 步：在我们的 Puppet 代理服务器上运行如下命令：

```
puppet agent -td
```

等一等。为什么使用 puppet agent 而不是 puppet apply 命令？agent 命令告诉代理执行 Puppet 主服务器为它保留的指令。apply 指令用于本地 puppet 清单的临时执行。

第一次运行 puppet agent 命令时，会出现证书错误，在代理上不会执行任何指令。这是因为 Puppet 主服务器必须授权 Puppet 代理服务器。在 Puppet 主服务器上，可以使用 puppet cert list 列出发出证书签署请求（CSR）的 Puppet 代理服务器列表。在生产环境中，应该选择授权的请求。但是，因为这只是一个测试环境，可以使用命令 puppet cert sign--all 签署所有

请求并继续工作。

在 Puppet 主服务器确认请求被处理之后，在 Puppet 代理服务器上再次运行 puppet agent -td 命令。从浏览器试着访问新的 Web 服务器，验证 Apache 正确安装和配置。

注意 使用带有 --noop 选项的命令（puppet agent--noop-td），可以预览 puppet agent 和 puppet apply 指令执行时做出的更改。noop 命令告诉 Puppet，你只想查看所做的更改，而不是真正地实施它们。

5.1.2 Hiera

Params.pp 很棒，但是，如果我们除了 CentOS 和 Ubuntu 还想考虑其他操作系统，会发生什么情况？我们必须更新 params.pp 中的条件语句。尽管 params.pp 相当短，但是必须考虑更大的数值列表和基于多种系统事实的多重条件语句。如果让 Puppet 确定在合适时机应用哪些数据，而不是自行编写更多的条件语句，不是更好吗？这就是 Hiera 的作用。

Hiera 是 Puppet 3.x 版本自带的一个开源工具。（要在更旧的 Puppet 版本上使用 Hiera，需要单独的人工安装。）Hiera 是代码与数据分离的一个流行备选方案，可以在各种数据存储上保存键 – 值数据，包括 YAML（YAML Ain't Markup Language，YAML 不是标记语言）、JSON（JavaScript 对象标记）文件和 MySQL 数据库。对于本书的 Hiera 示例，我们使用 YAML。

当我们想要引用 Hiera 中保存的值时，可以使用 hiera('*value*') 命令。如果在我们的 apache 模块中使用 Hiera，init.pp 清单如程序清单 5-5 所示。

程序清单 5-5 采用 Hiera 的 Apache 模块 init.pp

```
class apache {
  $webserver= hiera('webserver')
  $confpath = hiera('confpath')
  $htmlpath = hiera('htmlpath')

  package { 'apache':
        name   => $webserver,
        ensure => installed,
  }

  file {'apacheconf':
        name    => $confpath,
        ensure  => file,
        mode    => 600,
        source  => "puppet:///modules/apache/$webserver.conf",
        require => Package['apache'],
  }
```

```
  service {'apache':
      name      => $webserver,
      ensure    => running,
      enable    => true,
      subscribe => File['apacheconf'],
  }
  file {'apachecontent':
    name    => $htmlpath,
    ensure  => file,
    mode    => 644,
    content => template('apache/index.html.erb'),
    require => Service['apache'],
  }
}
```

我们来看看如何使系统做好使用 Hiera 的准备。首先，配置 Hiera 正确使用我们的数据。

Hiera 配置文件可以在 Puppet 企业版的 /etc/puppetlabs/puppet/hiera.yaml 中找到（在开源版 Puppet 中则是 /etc/puppet/hiera.yaml）。Hiera 的设置以 YAML 格式定义，这从配置文件的扩展名中可以猜出。

程序清单 5-6 展示了一个 Hiera 配置文件的样板。建立 hiera.yaml 文件的备份（如 hiera.yaml.old）。然后，更改 hiera.yaml 文件的内容，使其与程序清单 5-6 相同，要小心空格。

程序清单 5-6　hiera.yaml

```
---
:backends:
  - yaml
:hierarchy:
  - defaults
  - "%{::osfamily}"
  - "%{::fqdn}"
  - global

:yaml:
# datadir is empty here, so hiera uses its defaults:
# - /var/lib/hiera on *nix
# - %CommonAppData%\PuppetLabs\hiera\var on Windows
# When specifying a datadir, make sure the directory exists.
  :datadir: '/etc/puppetlabs/puppet/hieradata'
```

:backends: 段告诉 Hiera，键 – 值对将如何存储。同样，我们在实验室中使用 YAML。

:hierarchy: 段告诉 Hiera 在哪些文件中查找键 – 值对数据。这一段使用来自 facter（第 4 章已做介绍）的系统信息，用于根据操作系统家族建立 YAML 文件等情况。所以，

如果我们想要为基于 Debian 的操作系统（如 Ubuntu）使用特殊设置，可以将它们放在 Debian.yaml 文件中。（后面将看到一个这方面的例子。）Hiera 将根据配置文件这一段的顺序搜索值。

根据程序清单 5-6 中的 hiera.yaml 文件样板，如果你要求 Hiera 查找变量 $webserver，它将首先在 defaults.yaml 文件中查找。如果 defaults.yaml 没有 $webserver 的值或者该文件不存在，Hiera 将搜索名称匹配当前评估的代理服务器上 osfamily 值的 YAML 文件（例如，Debian.yaml）。这一过程将持续到 Hiera 找到所要搜索的值，或者在 global.yaml 文件中搜索完毕为止。如果在 global.yaml 中不存在该值，Hiera 将返回一个 nil 值给调用它的清单编译进程。

:yaml: 段告诉 Hiera 我们将在哪里存储所有在 :hierarchy: 段中指定的 YAML 数据文件。对于我们的 apache 模块，创建两个数据文件以匹配 %{osfamily}fact 变量返回的值（Red Hat、Fedora 和 CentOS 系统返回 RedHat，Debian 和 Ubuntu 系统返回 Debian）：Debian.yaml 和 RedHat.yaml。记住，要将这些文件放在你所指定的 :datadir: 路径（在例子中是 /etc/puppetlabs/puppet/hieradata）（参见程序清单 5-7）。

程序清单 5-7 操作系统特定 YAML 文件

```
Debian.yaml
---
webserver:
  - 'apache2'
confpath:
  - '/etc/apache2/apache2.conf'
htmlpath:
  - '/var/www/index.html'

RedHat.yaml
---
webserver:
  - 'httpd'
confpath:
  - '/etc/httpd/conf/httpd.conf'
htmlpath:
  - '/var/www/html/index.html'
```

仅为编写清晰代码而进行数据分离似乎微不足道，但是考虑一下另一个用例：管理密码等敏感数据。如果你将清单保存在一个存储库中供他人访问，可能不一定想让整个团队都能够访问关键资源的密码。在某个时点，该清单可能被没有授权以根权限访问重要资源（如数据库或者 DNS 服务器）的人所查看。即使你对 Hiera 文件有安全的访问控制措施，也可能应该用 hiera-gpg 等插件加密敏感数据，保证重要数据的安全。

5.1.3　节点分类

Hiera 还可以指定部署服务器所用的类。

回忆 Puppet 主服务器上 site.pp 中的如下条目：

```
node 'puppetnode01.devops.local' {
  include apache
}
```

我们可以将该条目更新如下：

```
node 'puppetnode01.devops.local' {
  hiera_include ('classes')
}
```

hiera_include 命令通知 Puppet 在 YAML 文件中搜索类定义。我们将在 /etc/puppetlabs/puppet/hieradata 目录中创建一个专用于该主机的 YAML 文件，该文件名为 puppetnode01.devops.local.yaml，包含如下条目：

```
---
classes:
  - apache
```

注意　你可能已经注意到，我们的所有 YAML 文件都在模块路径之外。这会使社区的其他成员难以使用我们的模块。Hiera 原创者 R.I.Pienaar 为这个问题开发了解决方案：开源项目 ripienaar/module_data。在你的模块中实现这一功能，就可以开发依赖于 Hiera 的可移植模块，但是这一主题超出了本书的范围。

5.2　应用层

在我们的部署中，应用层是一个简单的 PHP 服务器。我们将在设置中包含 PHP 二进制文件、与 MySQL 数据库通信的 PHP 组件和验证 PHP 成功运行的 index.php 样板文件。PHP 必须安装在与 Apache Web 服务器相同的系统上。所以，在我的设置中，将把 PHP 部署到 puppetnode01.devops.local。首先看看如何部署 PHP 组件，然后研究配置文件中需要进行的更改。

你可能已经猜到，模块的名称根据部署 PHP 的操作系统而不同。对于基于 Debian 的系统，模块名为 php5 和 php5-mysql。对于基于 Red Hat 的系统，模块名为 php 和 php-mysql。index.php 样板文件的路径和 Apache Web 服务器的 HTML 内容相同。我们将为 PHP 模块使用 Hiera，所以，将在 Debian.yaml 和 RedHat.yaml 文件中添加对应的值（参见程序清单 5-8）。

程序清单 5-8 添加到操作系统家族 Hiera 文件的 PHP 值

```
Debian.yaml
---
webserver:
  - 'apache2'
confpath:
  - '/etc/apache2/apache2.conf'
htmlpath:
  - '/var/www/index.html'
phpserver:
  - 'php5'
phpmysql:
  - 'php5-mysql'
phppath:
  - '/var/www/index.php'

RedHat.yaml
---
webserver:
  - 'httpd'
confpath:
  - '/etc/httpd/conf/httpd.conf'
htmlpath:
  - '/var/www/html/index.html'
phpserver:
  - 'php'
phpmysql:
  - 'php-mysql'
phppath:
  - '/var/www/html/index.php'
```

PHP 模块的最后一部分是在 Puppet 主服务器的模块路径下创建 module 目录（有关模块目录结构参见第 4 章），它包含对应的 init.pp 文件（参见程序清单 5-9）。

程序清单 5-9 PHP 模块的 init.pp 文件样板

```
class php {
  $webserver = hiera('webserver')
  package { 'php':
    name => hiera('phpserver'),
    ensure => installed,
  }
  package { 'phpmysql':
    name => hiera('phpmysql'),
    ensure => installed,
```

```
    require => Package['php'],
  }
  file { 'phpsample':
    name => hiera('phppath'),
    ensure => file,
    mode => 644,
    source => 'puppet:///modules/php/index.php',
    require => Package['phpmysql'],
  }
}
```

5.3　数据库层

多层应用程序的最后一个组件是数据库，在我们的部署中将使用 MySQL 数据库。尽管我们可以构建自己的模块，但是这里将利用 Puppet Forge 和 Puppet Labs 构建的 MySQL 模块。

在 Puppet 主服务器上，执行如下命令下载 MySQL 模块：

```
puppet module install puppetlabs/mysql
```

现在，我们编辑 site.pp，加入一个用于数据库服务器节点的条目。在我的实验室环境中，数据库服务器是 puppetnode02.devops.local：

```
node 'puppetnode02.devops.local' {
  hiera_include ('classes')
}
```

和 Web 服务器一样，我将创建一个专用于数据库服务器上部署的类的 YAML 文件（puppetnode02.devops.local.yaml），并设置 MySQL 数据库的根用户密码：

```
---
classes:
  - mysql::server

mysql::server::root_password: 'Puppetize1234'
```

上面的指令告诉数据库服务器部署 mysql::server 类，并根据我们指定的值设置数据库根密码。

5.4　实施生产建议措施

网络时间协议（NTP）对生产工作负载很关键，尤其是系统时钟可能随时广泛变化的虚拟机。即使只是测试实验室，我们也要应用好的系统管理员原则，在环境中部署 NTP。此外，因为时间同步在 VMware 单点登录（SSO）部署必不可少，所有 VMware 管理员现在都

很了解 NTP，对吗？

Puppet Forge 上有许多 NTP 模块（正如 MySQL 模块那样），但是，我们同样使用 Puppet Labs 开发的模块：

```
puppet module install puppetlabs/ntp
```

对于你的测试环境，使用指向 pool.ntp.org 的默认 NTP 服务器值即可。但是，在生产环境中，应该编辑模块的 params.pp 文件，使用环境中正确的 NTP 服务器。

因为 NTP 应该运行于环境中的所有服务器上，可以将其放在 Puppet 主服务器上 site.pp 文件的 defaults 段（class {'ntp':}），也可以将 NTP 类加到 YAML 文件中。

5.5 部署应用程序环境

在决定最喜欢的数据分离方法（params 类或者 Hiera）之后，相应地更新模块，并修改 site.pp 文件，使 puppetnode01 包含 php 及 apache 模块，puppetnode02 包含 mysql 模块。重新在 puppetnode01 和 puppetnode02 上运行 puppet agent 命令之后，将安装模块，并为构建应用程序做好准备。

5.6 小结

使用 Puppet 可以使 LAMP（Linux-Apache-MySQL-PHP）栈部署更简单。Puppet 模块可以利用数据分离，在单一位置更改 Puppet 代码使用的数据，从而实现可移植性。创建模块时，数据分离有两个选择：params 类和 Hiera。这两种方法都是有效的（但是 params 类方法似乎在 Puppet Forge 贡献者中更为流行）。在第 6 章，我们将研究 Puppet 与 VMware vSphere 的集成。

参考文献

[1] “Puppet 3 Reference Manual”: http://docs.puppetlabs.com/puppet/3/reference/

第 6 章 Chapter 6

用 Puppet 进行 VMware vSphere 管理

Puppet 可以用于管理 VMware vCloud 套件中的资源。Puppet 企业版有一个云配给组件，可以在 VMware vSphere 上部署新虚拟机。此外，VMware vCloud Air Team 已经开放了用于 vSphere 组件内部管理的一些模块源代码。对于本章中的例子，我们至少需要有一个在 vSphere 部署中创建的 Linux VM 模板，还需要一台 Puppet 企业版主服务器，它与连接到 vCenter 服务器实例的 vSphere 群集有网络连接。

本章包含如下主题：

- Puppet 的 VMware vSphere 云配给器
- VMware 的管理模块

6.1 Puppet 的 VMware vSphere 云配给器

系统管理员可以使用 Puppet 企业版云配给器在他们的环境中由模板部署 VM。云配给器是一个可选组件，所以，必须在 Puppet 企业版主服务器安装期间选择。

6.1.1 准备 VM 模板

确保在成为模板的 Linux VM 上安装 VMware Tools。验证 Linux VM 上的 Puppet 主服务器域名系统（DNS）条目是否正确。如果你工作于没有 DNS 服务器的测试实验室，也可以编辑模板 VM 的 /etc/hosts 文件，包含用于 Puppet 主服务器的条目。最后（但并非不重要），将你的安全外壳（SSH）公钥添加到 Linux 模板中的一个用户，最好是具备 sudo 权限的用户（为了易于管理）。

6.1.2 准备 Puppet 主服务器

确认 Puppet 企业版主服务器上安装了云配给器之后，必须在用于执行 vCenter 中管理任务的 Puppet 主服务器上配置证书。这些证书保存在 .fog 文件中，该文件保存于常规登录主服务器使用的用户主目录下。该用户不一定是根用户，但是为了简单起见，可以使用主服务器上的根用户。.fog 文件中的数据结构看起来应该很熟悉，因为它使用了 YAML 语法（参见程序清单 6-1）。

程序清单 6-1 .fog 文件样板

```
:default:
  vsphere_username: administrator
  vsphere_password: Puppetize1234
  vsphere_server: vCenter.devops.local
  vsphere_expected_pubkey_hash: 118d38f5267ca92948b9a8c9fdb8c64decf67530d4a
```

你可能觉得奇怪，“从哪里得到 .fog 文件的散列值？”是的，这是我们无法事先生成的。在配置 .fog 文件之后，需要对 vCenter 执行一个命令（例如，puppet node_vmware list），第一次执行时，会生成和显示这个散列值，让我们添加到 .fog 文件中。

现在，我们已经做好准备执行云配给器中的各种命令。在部署 VM 之前，必须知道有哪些模板可供部署。puppet node_vmware list 实现这一功能。在测试环境中，列出所有 VM 是可以的。但是，在生产环境中，可能需要过滤搜索结果。在那种情况下，可以将 list 命令通过管道送到 grep，只显示模板列表：puppet node_vmware list | grep true -B7。在例子中，我搜索关键字 true，因为有一个布尔值（template）表示列表中的 VM 是不是模板，如程序清单 6-2 所示。

程序清单 6-2 Puppet VM 列表

```
/Datacenters/VMTrooperHQ/vm/centos-6-x64-template
  powerstate: poweredOff
  name:       centos-6-x64-template
  hostname:   --------
  instanceid: 5005ed33-f57e-6106-36a7-b2ee1626bc4c
  ipaddress:  ---.---.---.---
  template:   true
--

/Datacenters/VMTrooperHQ/vm/precise-template
  powerstate: poweredOff
  name:       precise-template
  hostname:   --------
  instanceid: 50055b0a-b56b-6790-3295-fbc31e9608ca
  ipaddress:  ---.---.---.---
  template:   true
```

记下每个 VM 列表项的第一行，这实际上是 Puppet 访问模板的路径：

```
/Datacenters/VMTrooperHQ/vm/precise-template
```

- Datacenters：Puppet 搜索的 vCenter 数据中心对象。
- VMTrooperHQ：这个值对应于 vCenter Server 中数据中心对象的名称。
- vm：在 vCenter VM 和模板视图中，这是保存 VM 的根目录。如果你的 VM 保存在某个文件夹中，路径反映出这一点（例如，…/vm/Discovered virtual machine/…）。
- precise-template：VM 模板的名称。

接下来，我们将以所要部署 VM 的模板路径为参数，使用 create 命令。这个命令的两个重要输入是 list 输出的 VM 模板完整路径和在 vCenter 中为 VM 所取的名称。如果该命令是 shell 脚本的一部分，可能还要为它添加 wait 选项。脚本可以等待 VM 完成克隆，或者等待 VM 完成启动并成功获取 IP 地址。下面是 VM 创建的一个例子：

```
puppet node_vmware create--vmname=devops --
template=/Datacenters/VMTrooperHQ/vm/precise-template -i
```

--vmname 指定 VM 的名称。--template 指定 VM 模板路径。最后一个选项（-i）指定等待到 VM 启动并获取 IP 地址。如果希望脚本只等待 VM 克隆操作结束，将 i 改为 w。

在 VM 以某个 IP 地址（例中是 192.168.1.100）正常运行之后，Puppet 主服务器将用 install 命令在 VM 上部署一个代理：

```
puppet node install --keyfile=devops_rsa --login=root --installer-
payload=pe-precise.tar.gz --installer-
answers=answers.lastrun.puppetnode01.devops.local 192.168.1.100
```

注意，Puppet 部署由通用节点命令执行，而不是我们用过的 node_vmware。其他需要注意的项目如下：

- --keyfile：我们用于登录到 VM 的私钥。如果还没有完成这一步，可以在部署后的 VM 运行 ssh-keygen，生成新的公钥和私钥配对。
- --login：具备安装 Puppet 代理权限的用户。
- --installer-payload：第 4 章中下载的 Puppet 企业版安装文件 gzip 压缩包。
- --installer-answers：允许 Puppet 企业版安装程序部署代理而不提示各种选项的自动化输入。如果没有应答文件，Puppet 企业版安装媒体的 answers 目录中包含了示例。创建应答文件的进一步文档可以查询 Puppet Labs 网站：https://docs.puppetlabs.com/pe/latest/install_answer_file_reference.html。

想要删除 VM 时，使用 terminate 命令：

```
puppet node_vmware terminate /Datacenters/VMTrooperHQ/vm/devops
```

其他实用的 puppet node_vmware 命令包括 start 和 stop。如果在模板 VM 上安装了 VMware Tools，stop 命令将试图优雅地关闭 VM 操作系统。否则，可以使用 stop 命令的 force 选项关闭 VM 电源而无须等待。

6.2 VMware 的管理模块

除了云配给器，Puppet 用户可以利用 VMware vCloud Air 自动化团队与社区共享的模块。

- vmware/vcenter
- vmware/vcsa
- vmware/vshield
- vmware/vmware_lib

前 3 个模块实际执行 VMware 资源上的操作，第 4 个模块包含其他模块的支持代码。实际上，vmware/vmware_lib 在导入其他 3 个模块之一时导入。这些模块利用开源的 rbvmomi Ruby 应用程序编程接口（API）与 VMware 组件通信。

使用 vmware/vcenter 模块

本章的讨论集中在 vmware/vcenter 模块上。和第 4 章及第 5 章中使用的其他模块一样，一定要将 vmware/vcenter 模块目录放入主服务器的模块路径（用 puppet config print modulepath 验证）。程序清单 6-3 展示了使用 vcenter 模块的清单样板。

程序清单 6-3 vCenterTest.pp Puppet 清单

```
vcenter::host { 'vsphere01.vmtrooper.com':
  path => "/VMTrooperHQ",
  username => "root",
  password => "Puppetize1234",
  transport => Transport['vcenter'],
}
transport { 'vcenter':
  username => 'administrator',
  password => 'Puppetize1234',
  server   => 'vcenter.vmtrooper.com',
  options  => { 'insecure' => true }
}
```

我的清单使用 vmware/vcenter 模块中定义的 vcenter::host 类型，将 ESXi 主机添加到 vCenter。Puppet 定义类型是新概念，本书不做深入介绍。但是，我们花一点时间讨论它们究竟是什么。

当你观察我的清单中的 vcenter::host 定义类型实例时，语法应该很熟悉：和声明 Puppet 资源的语法相同。所以，Puppet 定义类型是用户定义自己的资源类型的一种方法。

你可能会想，“定义类型和类之间有什么不同？”是的，除了一个关键的差异之外，这两个概念很相似：Puppet 类设计为在清单中只使用一次，而 Puppet 定义类型可在清单中多次使用。

例如，在我们的 LAMP 部署中，只需要在 puppetnode01 服务器上部署一个 PHP 实例和

一个 Apache 实例。所以，类结构适合于我们创建的模块。当我们要在 vCenter 添加 ESXi 主机时，清单可能有多个 ESXi 主机定义，定义类型更适合于这类对象。

vcenter::host 定义类型有几个必需的参数。(我们将在稍后讨论可选参数。)

- path：主机放置的 vCenter 位置，通常是 vCenter 数据中心对象的根（例如，在我的环境中是 VMTrooperHQ）
- username：ESXi 主机的登录名
- password：ESXi 主机的密码
- transport：vCenter 连接

在继续之前，我们短暂地关注传输资源。options 参数可以告诉 Puppet 在连接到 vCenter 时忽略任何 SSL 错误。当然，在生产环境中不会使用这个选项。但是，对于测试实验室，我们可能没有正确的 SSL 证书。'insecure' => true 值启用忽略 SSL 错误的能力。在 vcenter::host 声明中，我们用 transport 元参数确保 Puppet 建立传输资源和 vcenter::host 之间的关系。

vcenter::host 定义类型相当灵活，如果希望设置附加选项（如使用的 NTP 服务器、启用 shell 或者 SSH 访问等），可以在清单中设置对应的选项。程序清单 6-4 详细说明了这些选项。

程序清单 6-4　vcenter::host Puppet 定义类型

```
# Copyright (C) 2013 VMware, Inc.
# Manage vcenter host resource
define vcenter::host (
  $path,
  $username,
  $password,
  $dateTimeConfig = {},
  $shells         = {},
  $servicesConfig = {},
  # transport is a metaparameter
) {

  $default_dt = {
    ntpConfig => {
      running => true,
      policy => 'automatic',
      server => [ '0.pool.ntp.org', '1.pool.ntp.org', ],
    },
    timeZone => {
      key => 'UTC',
    },
  }
  $config_dt = merge($default_dt, $dateTimeConfig)

  $default_shell = {
```

```
    esxi_shell => {
      running => false,
      policy => 'off',
    },
    ssh => {
      running => false,
      policy => 'off',
    },
    esxi_shell_time_out => 0,
    esxi_shell_interactive_time_out => 0,
    suppress_shell_warning => 0,
  }
  $config_shells = merge($default_shell, $shells)
  $default_svcs = {
    dcui => {
      running => true,
      policy => 'on',
    },
  }
  $config_svcs = merge($default_svcs, $servicesConfig)

  vc_host { $name:
    ensure    => present,
    path      => $path,
    username  => $username,
    password  => $password,
    transport => $transport,
  }

  # ntp
  esx_ntpconfig { $name:
    server    => $config_dt['ntpConfig']['server'],
    transport => $transport,
  }

  esx_service { "${name}:ntpd":
    policy    => $config_dt['ntpConfig']['policy'],
    running   => $config_dt['ntpConfig']['running'],
    subscribe => Esx_ntpconfig[$name],
  }

  # shells
  esx_shells { $name:
    # to disable cluster/host status warnings:
    #   http://kb.vmware.com/kb/2034841 ESXi 5.1 and related articles
```

```
    #   esxcli system settings advanced set -o /UserVars/
      SuppressShellWarning -i (0|1)
    #   vSphere API: advanced settings UserVars.SuppressShellWarning =
      (0|1) [type long]
    suppress_shell_warning => $config_shells['suppress_shell_warning'],
    # timeout means 'x minutes after enablement, disable new logins'
    #   vSphere API: advanced settings UserVars.ESXiShellTimeOut = [type
      long] (0 disables)
    #   http://kb.vmware.com/kb/2004746 ; timeout isn't 'log out user after
      x minutes inactivity'
    esxi_shell_time_out  => $config_shells['esxi_shell_time_out'],
    # interactiveTimeOut means 'log out user after x minutes inactivity'
    #   vSphere API: advanced settings UserVars.ESXiShellInteractiveTimeOut
      = [type long] (0 disables)
    esxi_shell_interactive_time_out  => $config_shells['esxi_shell_
      interactive_time_out'],
    transport => $transport,
  }

  esx_service { "${name}:TSM":
    policy    => $config_shells['esxi_shell']['policy'],
    running   => $config_shells['esxi_shell']['running'],
    subscribe => Esx_shells[$name],
  }
  esx_service { "${name}:TSM-SSH":
    policy    => $config_shells['ssh']['policy'],
    running   => $config_shells['ssh']['running'],
    subscribe => Esx_shells[$name],
  }

  # simple services
  # - fully managed by HostServiceSystem
  # - behaviors are boot-time enablement and running/stopped
  # - vSphere API provides no additional configuration
  esx_service { "${name}:DCUI":
    policy  => $config_svcs['dcui']['policy'],
    running => $config_svcs['dcui']['running'],
  }
}
```

在vcenter::host类型定义的开始，我们会看到空值赋值（"= {}"）的3个可选参数：

- dateTimeConfig：NTP服务器
- shells：关于shell\SSH访问启用、禁用的选项
- servicesConfig：ESXi主机上启用、禁用服务的选项

NTP服务器、shell选项和服务选项的默认值分别可以在$default_dt、$default_shell和

$default_svcs 条目中看到。你会注意到，其他地方定义了 esx_ntpconfig 和 esx_service 类型的实例，以执行必要的时间服务器和服务配置。

在清单中设置这些选项相当简单。程序清单 6-5 展示了启用 ESXi shell 和 SSH、抑制 vCenter 警告的样板清单。

程序清单 6-5 启用 ESXi Shell 和 SSH 的 vCenterTest.pp Puppet 清单

```
vcenter::host { 'vsphere01.vmtrooper.com':
  path => "/VMTrooperHQ",
  username => "root",
  password => "Puppetize1234",
  shells => {
    esxi_shell => {
      running => true,
      policy => 'on',
    },
    ssh => {
      running => true,
      policy => 'on',
    },
    esxi_shell_time_out => 0,
    esxi_shell_interactive_time_out => 0,
    suppress_shell_warning => 1,
  },
  servicesConfig => {
    dcui => {
      running => true,
      policy => 'on',
    },
    esxi_shell => {
      running => true,
      policy => 'on',
    },
    ssh => {
      running => true,
      policy => 'on',
    },
  },
  transport => Transport['vcenter'],
}

transport { 'vcenter':
  username => 'administrator',
  password => 'Puppetize1234',
  server   => 'vcenter.vmtrooper.com',
```

```
  options  => { 'insecure' => true }
 # options  => $vcenter['options'],
}
```

我用粗体强调添加到 venter::host 资源定义中的新段 shells 和 servicesConfig 中的自定义值。现在，我们似乎在 Puppet 资源内相互声明（Puppet 全面启动！），但是实际上不是如此。shell 和 servicesConfig 参数实际上以 Ruby 散列作为输入。

可以随意试验其他服务，在你有足够自信时甚至可以扩展代码支持新的功能。但是要记住，还可以试验其他两个 VMware 模块。

6.3　小结

Puppet 对 VMware 设施的支持为我们的 Puppet 工作流增加了 vSphere 组件管理的能力。现在，我们对 Puppet 及用于 VMware 技术的方法已经做了相当全面的介绍，在接下来的几章中将研究 Chef 配置管理系统。

参考文献

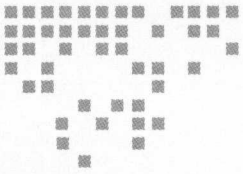
[1] “Puppet 3 Reference Manual”: http://docs.puppetlabs.com/puppet/3/reference/

[2] VMware Puppet Module documentation: https://forge.puppetlabs.com/vmware

第三部分 Part 3

Chef

- 第 7 章　Chef 简介
- 第 8 章　使用 Chef 完成系统管理任务
- 第 9 章　用 Chef 管理 VMware vSphere



Chapter 7 第 7 章

Chef 简介

本章介绍 Chef 的概念、核心思想、一些术语并研究 ChefDK 的功能，并编写你的第一个（简单）“食谱”（recipe）。在某些材料中，这些内容被分成数章进行介绍；但是由于 ChefDK 给 Chef 带来的飞跃，我们把 ChefDK 相关内容放在本章讲解。

本章包含如下主题：

- Chef 简介
- Chef 的核心思想
- Chef 术语
- 托管 Chef 或者 Chef Server
- ChefDK
- Knife
- 如何创建第一个“你好，世界”的 Chef 食谱

7.1 什么是 Chef

Chef 是 Adam Jacob 编写的基础设施自动化平台。其服务器部分用 Erlang 编写，客户端以 Ruby 编写并使用一种领域特定语言（DSL）。DSL 使 Chef 的新用户可以很快理解食谱（recipe）的编写方法。Chef 的目标是自动化基础设施，不管它们是私有云、公共云还是裸机。

用户编写食谱——描述机器预期状态的“烹调书”（cookbook）的一部分。不管你是要简单地管理服务器的网络时间协议（NTP）配置还是部署应用程序，都可以用 Chef 轻

松实现。食谱作者声明软件包、配置和与食谱关联的任何服务。为了保持数据驱动特性，烹调书使用属性。这可以从烹调书中的食谱里抽象出细节，使它们有很好的移植性和可重用性。

Chef 一度只是开放源码客户端和服务器的组合，但是近年推出的托管版本（软件即服务，SaaS）允许用户快速地采用 Chef 而无须维护 Chef Server 安装。2014 年，Chef（过去称作 Opscode）将其 Enterprise Chef 产品与开源产品合并，更名为 Chef Server。

Chef 可以用于管理 Linux、UNIX（Solaris、AIX 或 BSD 家族）及 Windows。

7.2　Chef 的核心思想

Chef 的开发是为了应对配置管理领域的需求（或者缺乏解决方案的情况）。在 Chef 发行时已经存在其他的工具和平台，但是每个工具都有不同的问题解决方案（或者思想）。Adam Jacob 以及所代表的 HJK Solutions 希望在 Chef 上重点解决的领域是食谱的顺序、幂等性、API 接口（实现搜索等功能）、端点处理（而不是由中央服务器进行编译）以及测试驱动基础设施。

7.2.1　食谱的顺序

编写食谱的顺序很重要。例如，安装 Tomcat 时，Tomcat 软件包必须在指定 server.xml 或 web.xml 配置之前安装。除此之外，Java 虚拟机（JVM），如 Oracle Java 或者 OpenJDK 必须存在，Tomcat 才能正常启动。食谱中的资源按照列出的顺序应用。还可以加入（或者“包含”）其他食谱，用户可以引用使应用程序正常运作的其他食谱（或者烹调书），创建描述整个应用的食谱。

7.2.2　幂等性

幂等性的思路是，食谱可以在机器上一次又一次地运行，结果应该始终相同。Chef 客户端检查系统，如果食谱中定义的资源没有变化，确保资源不会执行。例如，如果一个“软件包”资源用于定义 Apache 的安装，第二次运行食谱不会重新安装 Apache。

7.2.3　基于 API 的服务器

Chef Server 从一开始就被构建为可通过 HTTPS 协议访问的 REST 风格应用程序编程接口（API）服务器。REST API 提供了对 Chef Server 上对象的访问，包括节点、环境、角色、烹调书（及其版本）、食谱以及运行列表，通过节点或者用户之间的公 / 私钥交换管理。Chef Server（SaaS 或者自托管）的思路是作为数据（例如，食谱、烹调书、属性）的存储库，Chef Server 上存储的数据可以通过编写食谱中的搜索功能加以利用。

7.2.4 客户端进行所有搜集工作

基础设施自动化平台的大部分处理密集部分往往在于“编译”需要在执行 Chef 客户端的节点上进行的操作。Chef 客户端仅在需要时与 Chef Server（通过 API）交互，但是它会下载服务器上需要的所有烹调书、食谱、模板等。然后，Chef 客户端按照指定的顺序执行其运行列表（记住：顺序很重要！），在客户端运行成功后，将节点的所有状态存回 Chef Server。

7.2.5 测试驱动基础设施

使 Chef 如此强大的是 Chef 生态系统中的工具，这些工具验证和确保通过食谱定义的机器在投产之前进行了充分的测试。你不是在代码部署到生产环境之后才试图修复问题，而是早在食谱创建部分就修复了问题。利用越来越广泛采用的 Foodcritic、ChefSpec 和 Test Kitchen 等工具，Chef 于 2014 年向所有用户发布了 ChefDK（Chef Development Kit，Chef 开发工具包），提供了集成上述所有工具的一致开发环境。

7.3 Chef 术语

Chef 使用了许多和现代烹调术语相关的惯用语。随处可见的食谱、烹调书、食品评论（food critic）和试验厨房（test kitchen）等单词，让人似乎觉得 Chef 是难以使用的产品。本节为你解说这些术语。

7.3.1 食谱

食谱（recipe）是 Chef 最基本的配置元素。食谱用 Chef 的 DSL（以 Ruby 为基础）编写，是一组保存在烹调书中的资源和需求。编写食谱的思路是，你的目标应该是所要完成的是“什么”，而不是“如何”完成。

7.3.2 烹调书

烹调书（cookbook）是一组描述某个应用程序的食谱。烹调书包含描述应用程序状态的食谱、模板和一组默认属性，描述 nginx 的烹调书就是一个例子，其中包括描述 nginx 安装的食谱，以及 nginx 配置部分的其他食谱。

7.3.3 属性

属性是在食谱中用于提供正常默认设置或者覆盖其他设置的配置数据。属性可以在节点、角色、环境或者烹调书的级别上关联或者重新定义。

7.3.4 角色

角色是用户定义和创建的食谱和属性值集合，用于描述常见的配置。例子之一是必须应

用到包含系统软件包（如 ntp、sudoers 和 DNS 解析器等）的所有服务器的基本角色。

7.3.5　运行列表

运行列表（run list）是以定义的顺序（顺序很重要！）应用的食谱列表。运行列表可以包含 0 个或者多个角色或食谱。多个角色和食谱（以及角色中的角色）可能很有效果，但是要记住，顺序很重要。

7.3.6　资源

资源定义 Chef 管理下的节点上的单个配置项。在 Chef 客户端运行期间，评估每个资源。例如，食谱是一组资源，每个资源在节点上执行特定的操作。

7.3.7　环境

环境是一组有不同配置设置的系统，可以用于限制应该用于机器的烹调书版本，或者存在于不同环境之间的不同 IP 地址、用户名、防火墙配置。例如，可以考虑一个 Tomcat 烹调书，该烹调书的逻辑在生产环境及 QA 环境中的服务器上保持相同，但是在其 JDBC 配置中可能有一个不同的 IP 地址。

7.4　托管 Chef 和 Chef Server 之间的差别

使用 Chef Server 时，Chef 能够实现最大效能。Chef Server 有两种版本：Chef Software 提供的 SaaS——托管 Chef（Hosted Chef）和 Chef Server——由用户（而非 Chef Software）管理和使用的可安装程序。下面的几个小节解释这两种产品。

7.4.1　托管 Chef

Chef 初创时，它启动了一项 SaaS 服务，允许用户放弃在自己的数据中心安装 Chef Server 的需求。在纯粹的云环境下，客户不维护任何物理基础设施，而是将其全部托管到一个公共云（例如，EC2 或者 Azure），此时上述服务是一个很好的选项。默认情况下，Chef 允许托管 Chef 平台的所有用户使用 5 个免费节点，设置一个托管 Chef 账户只需要花费几分钟。（而且，更重要的是，不需要信用卡号码。）

7.4.2　Chef Server

Chef 也一直有 Chef Server 的开源版本。后来，它推出了 Enterprise Chef（也称为 Private Chef），该产品很类似于托管产品，但是运行于用户的数据中心，而不使用 Chef 的托管解决方案。2014 年，Chef 合并开源版本和 Enterprise Chef，称其为 Chef Server。Chef Server 允许

管理员在开源和企业版本之间切换，只需要单击一下鼠标（或者按下一个键）。

Chef Server 自带大量的功能，包括用户界面（UI）、基于角色的访问控制（RBAC）、多租户等开源功能，以及分析及报表等企业类功能。这些功能都添加到 Chef 的 API，你所编写的烹调书（或者运行的客户端）在 Chef Server 或者托管 Chef 上都能正常运行。


注意 在本书中，我们使用托管 Chef 基础设施，因为它的设置比较简单。

7.5 ChefDK 简介

ChefDK 是 Chef Software 发行的软件，包含了本书中练习所需的实用程序和工具。下面几个小节解释 ChefDK 的概念、安装方法以及给 Chef 带来的好处。

7.5.1 ChefDK 是什么

多年以来，Chef 不仅建立了关于烹调书的社区，还构建了测试工具（如 Kitchen CI）和烹调书依赖解决方案（如 Berkshelf）。这些工具大部分采用和 Chef 客户端相同的语言（Ruby）编写。Ruby 的最大问题是你所运行的 Ruby 版本，或者许多 Linux 分发版本编译和捆绑 Ruby 的方式。这可能导致工具的一致性问题。Chef 自带一个 Omnibus 软件包，其中有它自己编译的 Ruby 版本。这个版本可以与操作系统的 Ruby 版本并存，在 Chef 支持的所有平台及版本上保持一致的 Chef 客户端行为。

ChefDK 是一个应用软件包，将所有测试工具捆绑在一起，是 Chef 对为所有不同测试工具（如 Kitchen CI、ChefSpec 和 Foodcritic）提供一致环境的要求做出的回应。ChefDK 捆绑了所有相关模块（Ruby 和各种 Ruby 工具），在不同操作系统上提供这些软件包的相同版本。它还包含一个新工具以精简 Chef 工作流，此外还有 Chef 自带的所有普通工具（Chef 客户端、Knife 和 Ohai）。这样，用户可以确保在开发环境或者构建服务器（用于校验和测试验证烹调书内容的自动化任务）上的 Test Kitchen 版本是一致的。

7.5.2 安装 ChefDK

ChefDK 的安装极其简单。它支持所有主要的 Linux 变种、Mac OS X 10.8+ 和 Windows 7 或以上版本。在本书后面的例子中，我们使用 ChefDK 安装增强 Chef 的 vSphere Center 功能的附加插件。

按照如下步骤开始安装：

1. 进入 http://www.chef.io/downloads/chef-dk/，获取最新版本的 ChefDK 软件包，如图 7-1 所示。

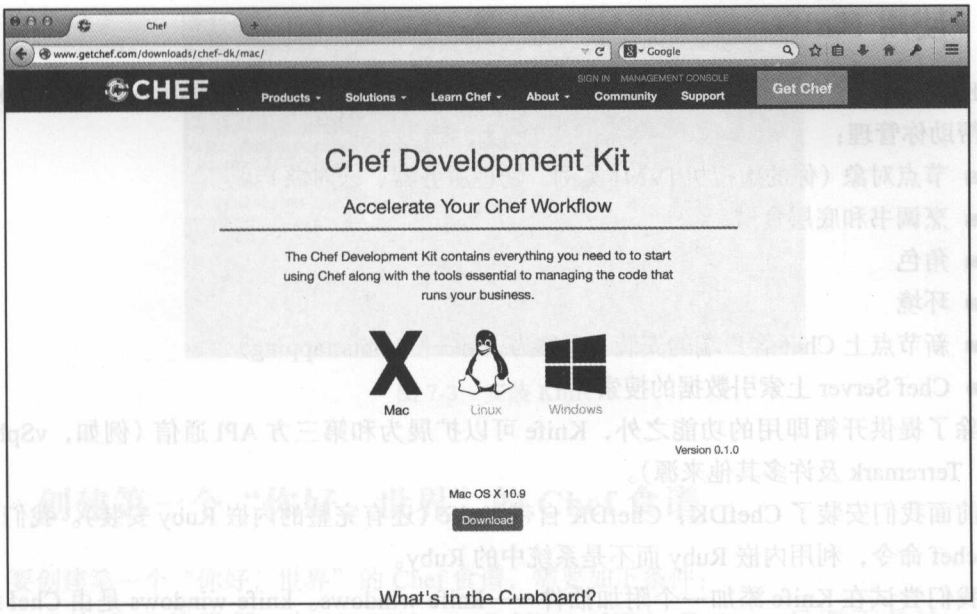


图 7-1　ChefDK 下载页面

Linux 或者 OS X 上的注意事项

如果在安装 ChefDK 之前已经安装了 Chef，ChefDK 安装程序将用这些工具的 ChefDK 安装版本替代 /usr/bin 中的默认符号链接（例如，/usr/bin/knife 将指向 /opt/chefdk/bin/knife）。

2. 打开终端 /PowerShell 窗口并运行：

```
chef verify
```

这将验证 ChefDK 的安装，应该显示“succeeded”（成功）：

```
$ chef verify
Running verification for component 'berkshelf'
Running verification for component 'test-kitchen'
Running verification for component 'chef-client'
Running verification for component 'chef-dk'
...
-----------------------------------------------
Verification of component 'berkshelf' succeeded.
Verification of component 'chef-dk' succeeded.
Verification of component 'chef-client' succeeded.
Verification of component 'test-kitchen' succeeded.
```

现在可以使用 ChefDK 了。

7.6 使用 Knife

Knife 是一个命令行工具，提供本地 Chef 存储库和 Chef Server 之间的一个接口。Knife 能够帮助你管理：

- 节点对象（你的虚拟机 [VM] 实例、物理服务器、云对象）
- 烹调书和底层食谱
- 角色
- 环境
- 新节点上 Chef 客户端的安装（也称为引导——bootstrapping）
- Chef Server 上索引数据的搜索

除了提供开箱即用的功能之外，Knife 可以扩展为和第三方 API 通信（例如，vSphere、EC2、Terremark 及许多其他来源）。

前面我们安装了 ChefDK，ChefDK 自带 Knife（还有完整的内嵌 Ruby 安装）。我们可以使用 chef 命令，利用内嵌 Ruby 而不是系统中的 Ruby。

我们尝试在 Knife 添加一个附加插件——knife-windows。knife-windows 是由 Chef 维护的可选 Knife 插件，为 Knife 提供 WinRM 功能。

打开一个终端 /PowerShell 窗口。

首先验证 knife-windows 没有安装（参见图 7-2）：

```
chef gem list knife-windows
```

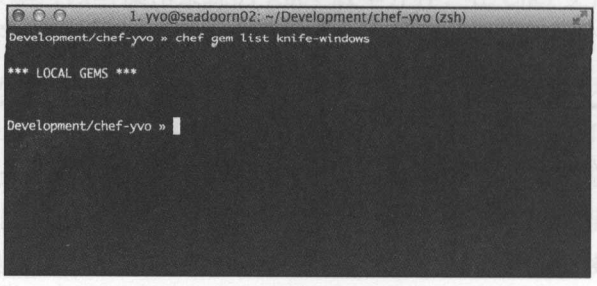


图 7-2 验证 Knife 是否安装

然后运行如下命令安装 knife-windows（参见图 7-3）：

```
chef gem install knife-windows
```

注意 根据 OS，输出可能稍有不同。gem 的作用是安装 knife-windows 和所有相关模块。

注意 Chef 在 http://docs.opscode.com/plugin_knife.html 维护一个当前 Knife 插件列表。

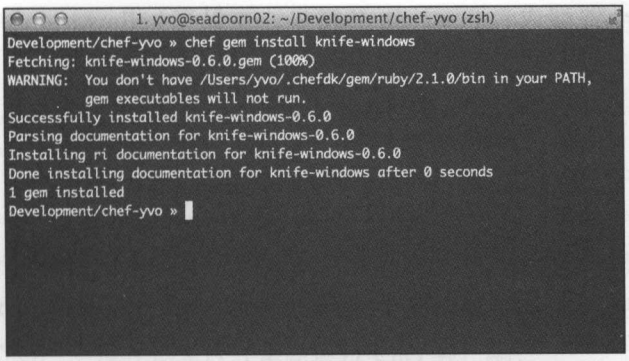


图 7-3　安装 Knife

7.7　创建第一个“你好，世界”的 Chef 食谱

要创建第一个“你好，世界”的 Chef 食谱，需要如下条件：

- 一个文本编辑器，如 Notepad ++、Sublime、VIM、Emacs，只要不是 Notepad 或 WordPad 就行
- 安装 ChefDK
- 一个空白的开发目录

在文本编辑器中，输入如下代码，并保存为 helloworld.rb：

```
# helloworld.rb - Creates a helloworld.txt in your home directory
file "#{ENV['HOME']}/helloworld.txt" do
        content "Hello World and welcome to Chef!\n"
end
```

保存文件之后，有些文本编辑器会发现它是 Ruby 文件，并相应地显示语法风格，如图 7-4 所示。

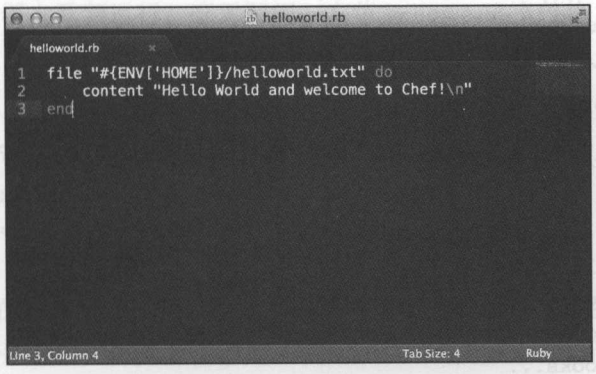


图 7-4　发现“你好，世界”代码是 Ruby 文件

现在，打开一个终端或者 PowerShell 窗口，在保存 helloworld.rb 的目录中执行如下命令：

```
$ chef-client --local-mode helloworld.rb
Starting Chef Client, version 11.14.0
resolving cookbooks for run list: []
Synchronizing Cookbooks:
Compiling Cookbooks...
Converging 1 resources
Recipe: @recipe_files::/Users/yvo/Development/chef-yvo/helloworld.rb
  * file[/Users/yvo/helloworld.txt] action create
    - create new file /Users/yvo/helloworld.txt
    - update content in file /Users/yvo/helloworld.txt from none to 1c6683
        --- /Users/yvo/helloworld.txt      2014-05-27 15:56:47.000000000
-0700
        +++ /var/folders/qz/zggg8hjd61b43hz4rl2twbcr0000gq/T/.helloworld.
txt20140527-7046-1fjplsq    2014-05-27 15:56:47.000000000 -0700
        @@ -1 +1,2 @@
        +Hello World and welcome to Chef!

Running handlers:
Running handlers complete
Chef Client finished, 1/1 resources updated in 1.533804 seconds
```

Chef 客户端结束时应该显示“1/1 resources updated。”这是因为该食谱运行一个资源——文件资源，以创建一个新文件。

验证该食谱执行了正确的操作。查看主目录，看看是否有一个名为 helloworld.txt 的文件，以及文件中保存的内容是否正确。

在 *NIX/OS X 终端窗口中，输入如下命令：

```
$ cat $HOME/helloworld.txt
    Hello World and welcome to Chef!
```

在 Windows PowerShell 窗口中，输入如下命令：

```
    > type $env:userprofile\helloworld.txt
  Hello World and welcome to Chef!
```

现在，我们在你的机器上重新运行 Chef 客户端，思路是输出将会反映 1 个资源中有 0 个被更新：

```
$ chef-client --local-mode helloworld.rb
Starting Chef Client, version 11.14.0
resolving cookbooks for run list: []
Synchronizing Cookbooks:
Compiling Cookbooks...
Converging 1 resources
```

```
Recipe: @recipe_files::/Users/yvo/Development/chef-yvo/helloworld.rb
  * file[/Users/yvo/helloworld.txt] action create (up to date)

Running handlers:
Running handlers complete

Chef Client finished, 0/1 resources updated in 1.457341 seconds
```

这说明了食谱的幂等性，Chef不需要进行任何更改，因为机器的预期状态没有变化。我们来做个更改，在文本编辑器中打开helloworld.txt文本，删除“Hello World and”部分，只留下“welcome to Chef!”。保存该文件并重新运行Chef客户端：

```
$ chef-client --local-mode helloworld.rb
Starting Chef Client, version 11.14.0
resolving cookbooks for run list: []
Synchronizing Cookbooks:
Compiling Cookbooks...
Converging 1 resources
Recipe: @recipe_files::/Users/yvo/Development/chef-yvo/helloworld.rb
  * file[/Users/yvo/helloworld.txt] action create
    - update content in file /Users/yvo/helloworld.txt from 821447 to
1c6683
        --- /Users/yvo/helloworld.txt      2014-05-27 16:05:13.000000000
-0700
        +++ /var/folders/qz/zggg8hjd61b43hz4rl2twbcr0000gq/T/.helloworld.
txt20140527-7230-1yj8vmq    2014-05-27 16:05:24.000000000 -0700
        @@ -1,2 +1,2 @@
        -welcome to Chef!
        +Hello World and welcome to Chef!

Running handlers:
Running handlers complete
Chef Client finished, 1/1 resources updated in 1.512766 seconds
```

Chef客户端在机器上重新运行，毫无疑问，它将helloworld文件恢复为预期的状态。

我们重新打开文本编辑器，创建另一个食谱。这个食谱清理helloworld所创建的“脏东西”。

在文本编辑器中，输入如下内容：

```
file "#{ENV['HOME']}/helloworld.txt" do
  action :delete
end
```

在helloworld.rb的同一目录保存该文件，并命名为cleanup.rb，如图7-5所示。

这段代码所做的是重用文件资源，但是不使用默认的创建操作，而是告诉食谱在遇到该文件时删除之。如果你试图进行应用程序部署，希望确保除了你所部署的版本之外不再维护

过去的版本，这样做就很有用。现在，以新创建的食谱运行 Chef 客户端：

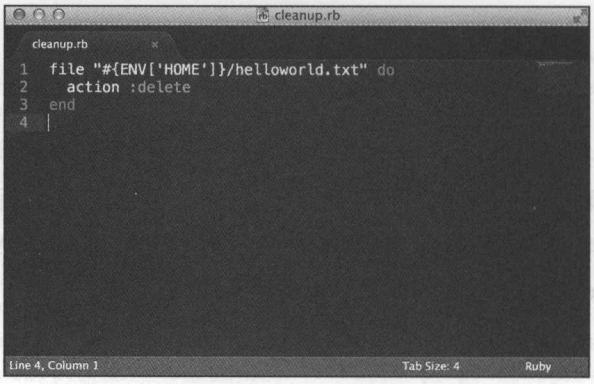


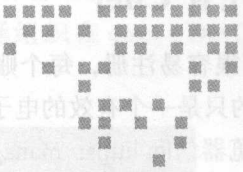
图 7-5 cleanup.rb

```
$ chef-client --local-mode cleanup.rb
Starting Chef Client, version 11.14.0
resolving cookbooks for run list: []
Synchronizing Cookbooks:
Compiling Cookbooks...
Converging 1 resources
Recipe: @recipe_files::/Users/yvo/Development/chef-yvo/cleanup.rb
  * file[/Users/yvo/helloworld.txt] action delete
    - delete file /Users/yvo/helloworld.txt
Running handlers:
Running handlers complete
Chef Client finished, 1/1 resources updated in 1.519316 seconds
Chef client
```

7.8 小结

本章介绍了 Chef 的历史、核心思想、术语和组成 Chef 生态系统的各种工具。此外，本章还介绍了 Chef Server 和托管 Chef 之间的不同之处。

你安装了 ChefDK，通过 ChefDK 安装了 Knife 插件，创建了头两个食谱，并且运行 Chef 客户端了解 Chef 与机器交互的方式。本书对 Chef 生态系统所做的介绍还很肤浅，但是你已经从术语的学习进入到最基本食谱的编写了，真了不起！



第 8 章 Chapter 8

使用 Chef 完成系统管理任务

有了新学的 Chef 知识，我们可以运用它们，维护 VMware 虚拟基础设施。本章介绍托管 Chef 的注册过程，解释社区食谱的概念，并浏览至少两个社区食谱的用例。本章还介绍 Knife 的更多功能，为第 9 章打好基础。

本章包含如下主题：

- 注册托管 Chef
- 社区食谱
- 设置系统管理
- 配置虚拟客户
- 实施策略
- 管理根密码

在开始之前，需要有如下条件：

- 一台运行 ChefDK 的工作站，可以访问 https://manage.chef.io
- 在 VMware 基础设施中有两个 Linux（推荐使用 RHEL/CentOS 6.5）虚拟机，启用安全外壳（SSH）
- 连接到机器时利用 sudo 获得根权限的能力
- 访问 http://www.chef.io 的能力
- 可以解压 zip 文件和 tag.gz 文件的程序（例如，7zip 或 WinRAR）
- 托管 Chef 账户（在本章中介绍）
- 可以访问 https://manage.chef.io 的 Web 浏览器（Firefox 20+、Google Chrome 20+ 或 IE 10+）

8.1 注册托管 Chef

托管 Chef 很容易注册，每个账户获得 5 个免费节点，在本书中，我们只使用其中的两个。注册所需的只是一个有效的电子邮件账户：

1. 打开浏览器访问 https://manage.chef.io，将会看到如图 8-1 所示的页面。

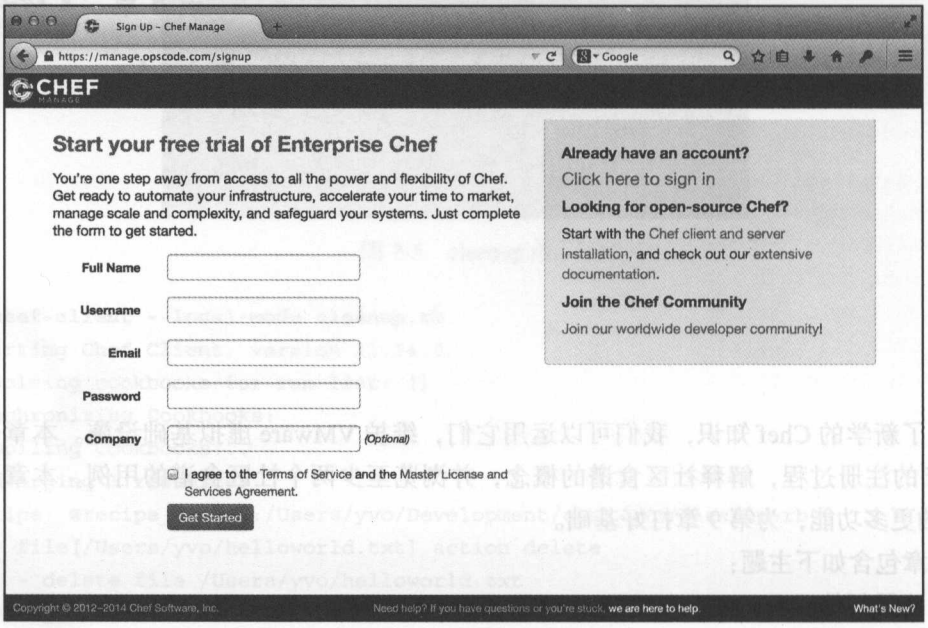


图 8-1 托管 Chef 首页

2. 填写文本字段以注册账户，如图 8-2 所示。

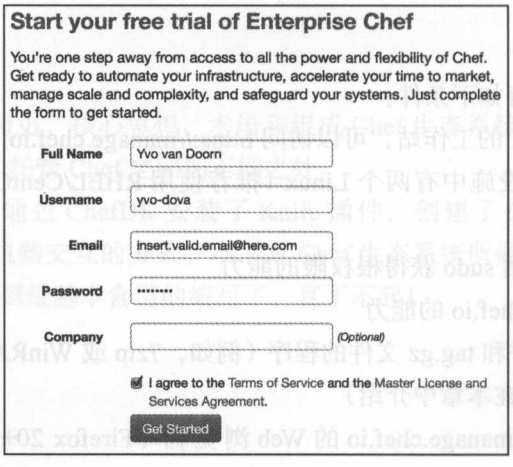


图 8-2 注册 Chef 账户

3. 单击 Get Started 之后，会看到加入现有组织或者创建新组织的提示，如图 8-3 所示。对于本练习，我们创建一个新组织。你可以选择组织在 dova（DevOps for VMware administrators）中的名称。

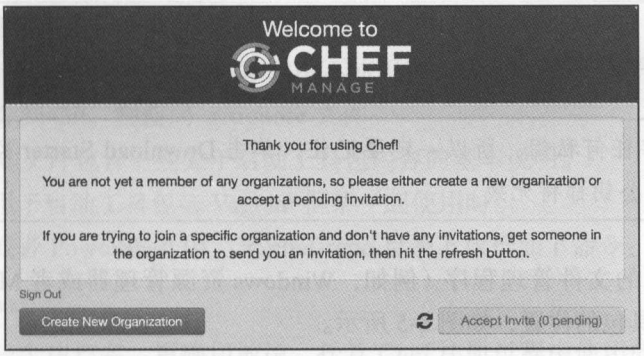


图 8-3　加入 / 创建 Chef 组织

单击 Create New Organization（创建新组织），托管 Chef 显示如图 8-4 所示的屏幕，以创建新的组织。

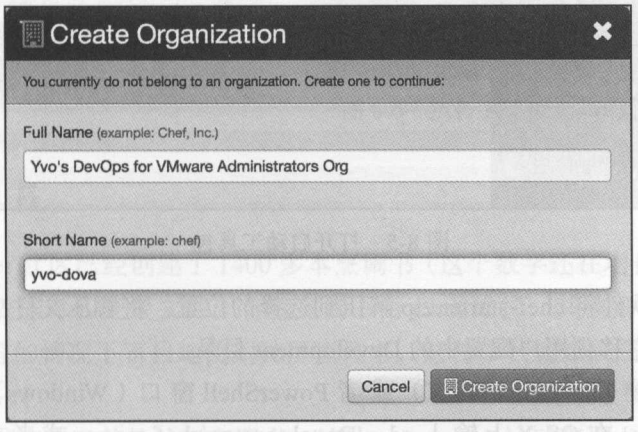


图 8-4　在托管 Chef 上创建新组织

创建组织之后，需要花费一些时间在 Chef 的各个服务器上创建新组织。以后，你将在 Chef 网站的 Getting Started 部分登录。

用 Starter Kit 设置本地存储库

在托管 Chef 平台上创建用户及组织之后，我们将设置你的工作站。工作站用于通过 HTTPS 与托管的 Chef Server 通信。设置工作站有两种途径：使用启动工具包（starter kit）或者人工设置。对于本练习，我们使用启动工具包加速工作站设置。

确认要配置为工作站的机器上安装了 ChefDK。可以分发启动工具包 zip 文件，设置任何数量的工作站。记住，启动工具包包含一个私钥，允许人们与你的托管 Chef 账户通信，所以一定要保存在安全的位置。

1. 单击 Download Starter Kit。

Chef 不保存任何私钥，所以一定要记住，单击 Download Starter Kit 时，用户和组织账户之前的公钥都将无效。

2. 在操作系统的文件管理程序（例如，Windows 资源管理器或者 Mac OS X 的 Finder）中打开下载启动工具包的位置，如图 8-5 所示。

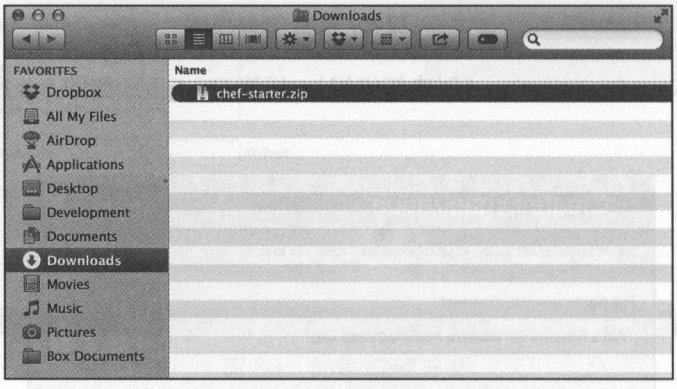


图 8-5 打开启动工具包

3. 用 zip 解压软件将 chef-starter.zip 解压到选择的位置。将解压文件夹命名为 chef-repo。在本书中，我将把它移到用户配置中的 Development 目录。

4. 打开终端窗口（Linux/OS X）或者 PowerShell 窗口（Windows），进入新的 chef-repo 文件夹（例如，在 OS X 上输入 cd ~/Development/chef-repo，或者在 Windows 上输入 cd$env:userprofile\Development\chef-repo）。执行 ls -al（在 OS X/Linux/UNIX 上）或者 dir -Force -File（在 Windows 上），列出文件目录，应该报告至少 3 个目录和 3 个文件：

```
$ ls -al
total 24
drwxr-xr-x@  8 yvo  staff   272 May 31 14:54 .
drwxr-xr-x  29 yvo  staff   986 May 31 14:54 ..
drwxr-xr-x@  5 yvo  staff   170 May 31 14:54 .chef
-rw-r--r--@  1 yvo  staff   495 May 31  2014 .gitignore
-rw-r--r--@  1 yvo  staff  2416 May 31  2014 README.md
-rw-r--r--@  1 yvo  staff  3649 May 31  2014 Vagrantfile
```

```
drwxr-xr-x@  4 yvo  staff   136 May 31  2014 cookbooks
drwxr-xr-x@  3 yvo  staff   102 May 31 14:54 roles
$ "
```

- .chef 目录是客户端配置和私钥的保存位置。
- cookbooks 目录包含一个烹调书的样板，在本章中不使用。
- roles 目录包含角色文件样板，在本章中不使用。
- 如果计划保存到 Git，则使用 .gitignore 文件。
- README.md 文件包含一些关于启动工具包内容的基本说明。
- Vagrantfile 用于启动工具包与 Vagrant 框架一起使用时。

5. 在终端窗口或者 PowerShell 中，于 chef-repo 目录下执行如下命令：

```
knife user list
```

上述命令返回一个用户名，更确切地说，托管 Chef 注册过程中使用的用户名，如图 8-6 所示。

执行 Knife 时，它总是试图定位一个 .chef 目录，该目录包含存储 Knife 配置的 knife.rb 文件。这就是在本练习中必须切换到 chef-repo 目录的原因。如果试图在 chef-repo 目录之外执行 Knife，会返回错误。

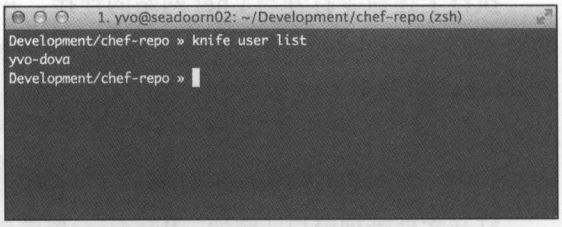


图 8-6　执行 Knife 以验证托管 Chef 的连接性

8.2　社区烹调书

多年以来，Chef 社区已经创建了 1400 多本烹调书（这个数字还在增长），这些烹调书可以管理应用程序、数据库和配置文件。有些烹调书由 Chef 维护，其他则由社区维护。不要在没有审核食谱内容的情况下盲目地使用社区烹调书，这种审核就像试图在厨房中创新菜肴时查看实际的作料一样。

有些社区烹调书依赖于其他社区烹调书。（例如，apache2 社区烹调书依靠 logrotate 烹调书设置合适的日志文件轮换。）你可以使用 Berkshelf（ChefDK 的一部分）程序解析烹调书之间的相互依赖。

8.3　设置系统管理

围绕配置管理的早期实现之一涉及网络时间协议（NTP）客户端和服务配置等基本系统服务的管理、密码管理和其他基本任务（与自动化应用交付或者数据库模式更新等相比）。在本节中，我们使用 NTP 社区烹调书，对 VMware 虚拟机（VM）应用策略。

8.3.1 准备 / 设置系统管理任务 1：管理时间

本章介绍的第一个系统管理活动与时间管理相关。VMware ESXi 提供了通过 VMware Tools 管理时间的潜力，但是 VMware 建议用 NTP 维护客户机的时间。系统管理任务的第一部分是进行某些准备工作，包括下载 NTP 烹调书并上传到托管 Chef 组织。

在 Chef 社区网站发布的 NTP 烹调书提供了一个食谱，可以启用 NTP 客户端。默认情况下，这个食谱设置 NTP 客户端配置，与默认 NTP 池服务器通信。我们将通过一个环境属性修改服务器列表。

下载 NTP 烹调书

打开终端或者 PowerShell 窗口，转到本章前面解压的 chef-repo 目录。

发出如下命令：

```
knife cookbook site download ntp
```

执行上述命令打开与 Chef 网站的连接，并下载指定的 NTP 烹调书。你会看到类似如下的输出：

```
$ knife cookbook site download ntp
Downloading ntp from the cookbooks site at version 1.6.2 to /Users/yvo/
Development/chef-repo/ntp-1.6.2.tar.gz
Cookbook saved: /Users/yvo/Development/chef-repo/ntp-1.6.2.tar.gz
```

打开文件管理窗口（例如，Windows 资源管理器或者 OS X 的 Finder），进入下载的 NTP 烹调书所在目录。在可以解压 tar.gz 文件的程序中打开该文件并提取内容（应该是一个 ntp 文件夹），保存到 chef-repo 目录中的 cookbooks 目录。

cookbooks 目录是维护计划用于 Chef 的所有烹调书的位置。

从 Chef 网站下载的 tar.gz 文件提取内容之后，可以从 chef-repo 目录中删除该文件。

在终端或者 PowerShell 窗口中，于 chef-repo 目录下发出如下命令：

```
knife cookbook upload ntp
```

上述命令将 NTP 烹调书上传到 Chef 服务器上你的托管 Chef 存储库，你将看到类似如下的输出：

```
$ knife cookbook upload ntp
Uploading ntp          [1.6.2]
Uploaded 1 cookbook.
```

可以用两种不同的方式确认上传：

1. 通过命令行，发出 knife cookbook list ntp 命令，该命令将返回上传的 NTP 烹调书版本。命令的输出如下：

```
$ knife cookbook list ntp
ntp   1.6.2
```

2. 通过 GUI，打开一个 Web 浏览器并进入 https://manage.chef.io，用本章前面创建的凭据登录。单击界面顶端的 Policy（策略），默认情况下，这将带你进入 Cookbooks 部分，在 GUI 主区域将会看到 NTP 烹调书，如图 8-7 所示。

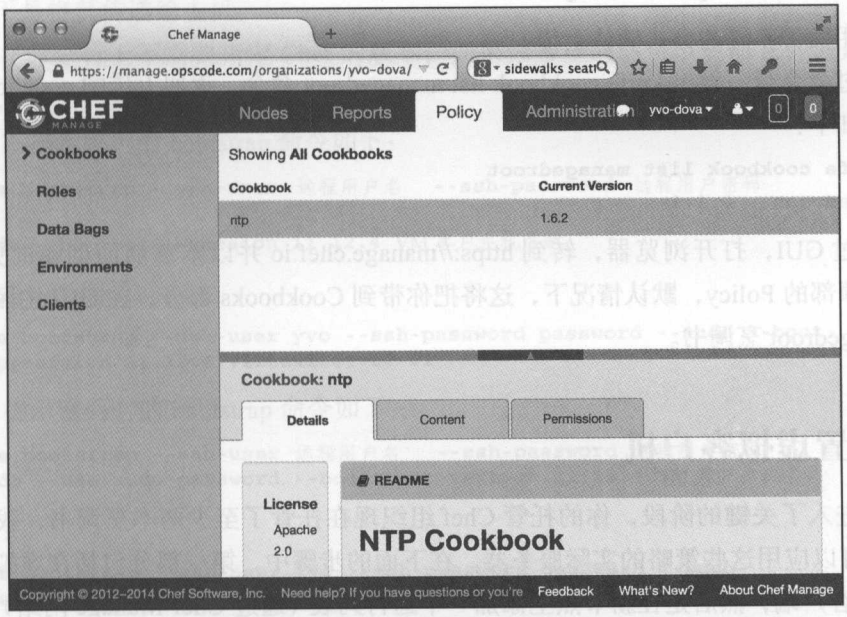


图 8-7　确认 NTP 烹调书上传

8.3.2　准备 / 设置系统管理任务 2：管理根密码

本章介绍的第二个系统管理任务是管理两个测试 VM 的根密码。这里使用的烹调书是专为本书编写的。注意，虽然这本烹调书的工作正如预期，但是 Chef 提供的根目录散列存储方法比属性更安全。

我们将使用管理密码散列的属性，并在不同的 Chef 环境中更改它，以说明 Chef 设置和处理属性的方式。

下载 Manage Root 烹调书

在 GitHub 上的一个存储库中维护的管理根密码烹调书是专为本书创建的。打开浏览器输入：http://bit.ly/1kwP34o。这个 URL 相当长，所以对其简写。

从这个网站，下载一个 zip 文件到浏览器的下载文件夹。打开操作系统的文件管理器工具，并转到浏览器的下载文件夹。将下载的文件（yvovandoorn-managedroot.zip）解压到我们在托管 Chef 设置期间建立的 chef-repo 目录下的 cookbooks 目录。

确保解压的 zip 文件中的目录命名为 managedroot 而不是 yvovandoorn-managedroot。如果命名的方式不是这样，则将其更名为 managedroot。

在终端或者 PowerShell 窗口中，于 chef-repo 目录下发出如下命令：

```
knife cookbook upload managedroot
```

可以用两种不同的方式确认上传：

1. 通过命令行，发出 knife cookbook list managedroot 命令，返回上传的 cookbook 版本。输出如下：

```
$ knife cookbook list managedroot
managedroot   0.1.1
```

2. 通过 GUI，打开浏览器，转到 https://manage.chef.io 并以本章前面创建的凭据登录。单击界面顶部的 Policy，默认情况下，这将把你带到 Cookbooks 部分，在 GUI 主区域中可以看到 managedroot 烹调书。

8.4 配置虚拟客户机

现在进入了关键的阶段，你的托管 Chef 组织现在托管了至少两本烹调书，我们所需要的是一些可以应用这些策略的实际服务器。在下面的步骤中，第一部分包括在虚拟客户机上安装 Chef 客户端，然后是在新节点上添加一个运行列表（通过 Chef Manage 网站）并以新的策略列表重新运行 Chef 客户端。

安装 Chef 客户端

Knife 可以启动一个节点。在现有版本的 Chef 中，Knife 可以启动接受安全外壳（SSH）连接或者 WinRM 连接的节点。我们将使用两个虚拟客户机启动及配置 Chef。

使用的服务器名是 virtualserver-01 和 virtualserver-02。我们从第一个服务器的启动开始（在下面的例子中是 virtualserver-01）。

确保虚拟客户机有能力连接到 https://www.chef.io。在你的 VM 通过 NAT 网关或者在私有网络上时，确保它们能够连接到互联网。

我们将要执行管理时间的系统管理任务，尽管有些自相矛盾，但是在执行引导命令之前，要确保虚拟客户机时钟的偏移不超过 15 分钟。如果偏移超出范围，可以在虚拟客户机上执行 sudo ntpdate 0.pool.ntp.org 命令，正确设置时间。

打开一个终端或者 Windows PowerShell 终端，转到 chef-repo 目录。这里执行的命令根据虚拟客户机的安装方式而略有不同。如果在虚拟客户机上的用户账户有不提示密码的 sudo 权限，使用的命令和需要输入密码时使用的命令略有差异。在下面的例子中，virtualserver-01 不需要密码，但是 virtualserver-02 需要密码。

可以选择在命令行上省略 --ssh-password 部分。如果这样做，将提示在输入密码时按下回车键而不是将其传递给主机。

虽然在命令行上不需要指定 Chef 的版本，如果你希望的话可以忽略它，但是我们在此加入版本，确保顺畅的学习体验。

sudo 不提示密码的 bootstrap 命令如下：

```
knife bootstrap --ssh-user 远程用户名  --ssh-password 远程用户密码
--sudo --bootstrap-version 11.12.4 VM 客户主机名称
```

下面是一个例子：

```
knife bootstrap --ssh-user yvo --ssh-password password --sudo --boot-
strap-version 11.12.4 virtualserver-01
```

sudo 提示密码时的 bootstrap 命令如下：

```
knife bootstrap --ssh-user 远程用户名  --ssh-password 远程用户密码
--sudo --use-sudo-password --bootstrap-version 11.12.4 VM 客户主机名称
```

例如：

```
knife bootstrap --ssh-user yvo --ssh-password password --sudo
--use-sudo-password --bootstrap-version 11.12.4 virtualserver-02
```

下面是 bootstrap 命令输出的删节版本：

```
chef-repo/cookbooks " knife bootstrap --ssh-user yvo --ssh-password
password --sudo --bootstrap-version 11.12.4 virtualserver-01
Connecting to virtualserver-01
virtualserver-01 Installing Chef Client...
....
virtualserver-01 Installing Chef 11.12.4
virtualserver-01 installing with rpm...
virtualserver-01 warning: /tmp/install.sh.1295/chef-11.12.4-1.el6.x86_64.
rpm: Header V4 DSA/SHA1 Signature, key ID 83ef826a: NOKEY
virtualserver-01 Preparing...                ##################################
############## [100%]
virtualserver-01    1:chef                   ##################################
############## [100%]
virtualserver-01 Thank you for installing Chef!
virtualserver-01 Starting first Chef Client run...
....
virtualserver-01 Starting Chef Client, version 11.12.4
virtualserver-01 Creating a new client identity for virtualserver-01.chef-
demo.com using the validator key.
```

```
virtualserver-01 resolving cookbooks for run list: []
virtualserver-01 Synchronizing Cookbooks:
virtualserver-01 Compiling Cookbooks...
virtualserver-01 Converging 0 resources
virtualserver-01
virtualserver-01 Running handlers:
virtualserver-01 Running handlers complete
virtualserver-01
virtualserver-01 Chef Client finished, 0/0 resources updated in 6.202156227
seconds
```

上述输出来自 virtualserver-01。因为我们没有为新启动的节点指定任何策略（这可以在引导时完成），所以应该看到“0/0 resources updated。”

重新运行 bootstrap 命令，现在指向第二个虚拟客户机。在两个虚拟客户机都已经引导（安装了 Chef 客户端）之后，在 chef-repo 目录下，于终端或者 Windows PowerShell 窗口运行如下命令：

```
knife node list
```

上述命令应该返回现在注册到托管 Chef 组织的两台服务器：

```
$ knife node list
virtualserver-01.chefdemo.com
virtualserver-02.chefdemo.com
$
```

8.5 系统管理任务

现在，两台服务器（或者按照 Chef 的说法——节点）已经配置，我们可以将 NTP 烹调书添加到服务器的一个运行列表中，以便在下次 Chef 客户端在服务器上执行时，运行必要的步骤使服务器实现相容性。

仅为两台注册服务器之一执行如下的步骤。

1. 打开浏览器并进入 https://manage.chef.io。

2. 单击界面顶端的 Nodes 文本，应该显示托管 Chef 中注册的两台服务器，如图 8-8 所示。

3. 单击第一台服务器，界面下半部分将显示各种选项。找到 Run List（运行列表）部分，现在该部分应该是空白的。单击 Edit（编辑）继续，如图 8-9 所示。

4. 显示新界面，可以在其中编辑节点的运行列表。界面中将显示所有已经上传到 Chef 组织的可用食谱。现在，只在当前运行列表中添加 NTP，在 Available Recipes（可用食谱）下选择 ntp 并拖入 Current Run List（当前运行列表），如图 8-10 所示。

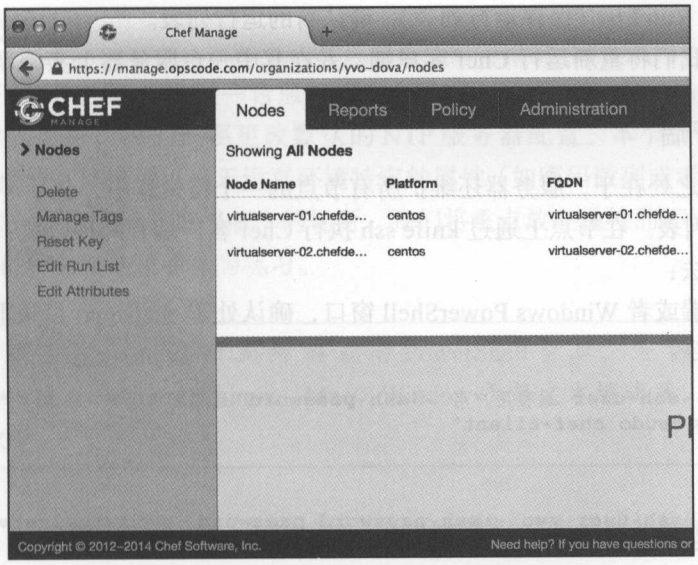


图 8-8　显示托管 Chef 中注册的服务器

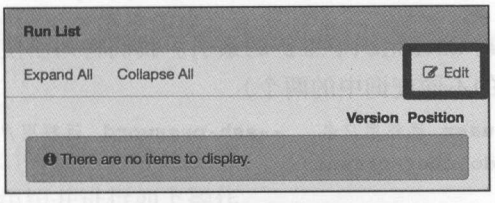


图 8-9　修改运行列表

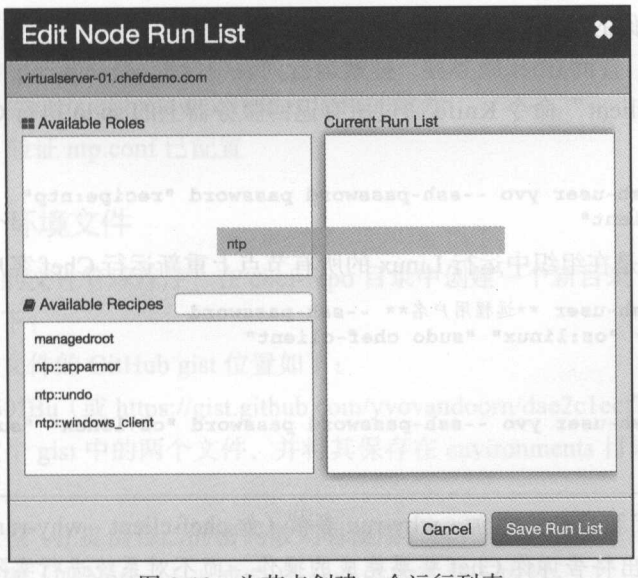


图 8-10　为节点创建一个运行列表

5. 单击 Save Run List（保存运行列表），确认新的运行列表。

下一小节，我们将重新运行 Chef 客户端，并在其中一台服务器上实施新的运行列表。

运行 Chef 客户端

Chef 的强大之处在于，服务器还维护所有节点的一个搜索索引，包括环境中所有节点上当前设置的运行列表。在节点上通过 knife ssh 执行 Chef 客户端有两种方法：

1. 麻烦的方法：

打开一个终端或者 Windows PowerShell 窗口，确认处于 chef-repo 目录下。

执行如下命令：

```
knife ssh --ssh-user 远程用户名 --ssh-password 远程用户密码 -m
VM客户主机名称 "sudo chef-client"
```

例如：

```
knife ssh --ssh-user yvo --ssh-password password -m virtualserver-01
"sudo chef-client"
```

这将专门针对某一个虚拟客户机，而不使用搜索索引。

2. 搜索方法：

也可以使用 Knife 搜索 Chef 组织中维护的索引。我们将列出使用搜索的两种方法（这是许多可用于远程执行命令的不同查询中的两个）：

a）`knife ssh --ssh-user 远程用户名 --ssh-password 远程用户密码 "recipe:ntp" "sudo chef-client"`

我们将其分解：

- ssh-user 是你的用户名。
- password 是你的密码。
- “recipe:ntp”查询 Chef 服务器，搜索运行列表中设置了 recipe:ntp 的所有节点。
- “sudo chef-client”命令 Knife 通过所有返回服务器上的 sudo 执行 Chef 客户端。

例如：

```
knife ssh --ssh-user yvo --ssh-password password "recipe:ntp"
"sudo chef-client"
```

b）另一种方法是在组织中运行 Linux 的所有节点上重新运行 Chef 客户端：

```
knife ssh --ssh-user **远程用户名** --ssh-password
**远程用户密码** "os:linux" "sudo chef-client"
```

例如：

```
knife ssh --ssh-user yvo --ssh-password password "os:linux" "sudo
chef-client"
```


注意 可以在 Chef 客户端上添加 --why-run 参数（如 chef-client --why-run），模拟 Chef 客户端运行。输出将告诉你 Chef 将要完成的操作，而不对系统进行实际更改。

8.6　管理根密码

现在我们正在用烹调书管理一台服务器的 NTP 配置，下面对 managedroot 烹调书做同样的操作。除此之外，我们还要更改默认的 NTP 服务器配置。本节介绍 Chef 中的环境（environments）功能，环境可以用于设置环境特定的属性（如密码散列或者唯一的 NTP 服务器）；它们也是限制所应安装烹调书版本的场所。我们将重点放在独特的环境特定属性，而将烹调书版本限制的探索留给读者作为练习。

注意　在这个练习中，我们回到终端或者 PowerShell 窗口，还需要 Windows 上的 Notepad++ 或者 Sublime（Linux、Mac、Windows）等文本编辑器，因为我们将创建几个新文件。

我们创建了几个密码散列样板（SHA-512，在 CentOS 6.5 上生成）并保存在一个 GitHub gist 上：

http://bit.ly/SnkO4H（或 https://gist.github.com/yvovandoorn/083f44dff822c168b971）

我们将对生产和测试 /QA 环境使用保存在这个 gist 中的散列。

我们即将完成如下工作：

- 创建两个环境文件
- 将每个环境文件上传到托管 Chef 组织
- 登录到托管 Chef 组织并进行如下操作
  - 为每台服务器分配一个环境
  - 修改每台服务器的运行列表，以运行 managedroot 烹调书
- 使用 search 从命令行重新运行 Chef 客户端，对服务器应用策略
- 用 knife ssh 验证 /etc/shadow 文件显示应用的散列
- 用 knife ssh 验证 ntp.conf 已配置

8.6.1　创建两个环境文件

使用操作系统的文件管理程序，在 chef-repo 目录中创建一个新目录 environments。

建议的路径如下：

包含两个环境文件的 GitHub gist 位置如下：

http://bit.ly/U60JBu（或 https://gist.github.com/yvovandoorn/dae2c1ecf71452bf8057）

你可以下载这个 gist 中的两个文件，并将其保存在 environments 目录中，该目录创建于 chef-repo 根目录下。

手工输入的途径（在提供自己的密码散列时使用）如下：

打开文本编辑器（如 Notepad++ 或 Sublime）并输入如下内容：

```
Content:
name "production"
description "Your production environment"
default_attributes ({
        "root" => {
                "password" => "$6$uLD5kuid$pvYSgFZMp5a.m8Q2fjVaIhcCIyhG-
FytWZYQXCkCW8oC.JvNWV5hmCptApdBOBZLA4DJc.rimnngPF9oZoZ6Ga1"
        },
        "ntp" => {
                "servers" => [ "0.vmware.pool.ntp.org", "1.vmware.pool.ntp.
org", "2.vmware.pool.ntp.org" ]
        }
})
```

将该文件保存在 chef-repo 目录下的 environments 目录，命名为 production.rb。

打开新文本编辑器窗口，输入如下内容：

```
name "testqa"
description "Your testqa environment"
default_attributes ({
        "root" => {
                "password" => "$6$EYgMQC0o$vaFIm/f81YRTjihvIk5brpE4oJhSiPn4Y-
2Bw2hsxOkC4og4V7KfSCYEmkw40MriUHICZppkGwPpZ4r44tGCIF1"
        },
        "ntp" => {
                "servers" => [ "0.pool.ntp.org", "1.pool.ntp.org", "2.pool.
ntp.org" ]
        }
})
```

将该文件保存在 chef-repo 目录下的 environments 目录，命名为 testqa.rb。

现在，你已经创建或者下载了两个环境文件并将其保存在 chef-repo 目录下的 environments 目录。

8.6.2 将环境文件上传到托管 Chef 组织

在终端或者 Windows Powershell 窗口中，转到 chef-repo 目录。

执行如下命令：

```
knife environment from file production.rb testqa.rb
```

将会看到类似如下的输出：

```
$ knife environment from file production.rb testqa.rb
Updated Environment production
Updated Environment testqa
```

执行如下命令确认两个新环境的存在：

```
knife environment show production
```

这条命令返回类似于图 8-11 所示的输出。

```
2. yvo@seadoorn02: ~/Development/chef-repo (zsh)
Development/chef-repo » knife environment show production
chef_type:          environment
cookbook_versions:
default_attributes:
  ntp:
    servers:
      0.vmware.pool.ntp.org
      1.vmware.pool.ntp.org
      2.vmware.pool.ntp.org
  root:
    password: $6$uLD5kuid$pvYSgFZMp5a.m8Q2fjVaIhcCIyhGFytWZYQXCkCW8oC.JvNWV5hmCptApdBOBZLA4DJc.rimnngPF9oZoZ6Ga1
description:        Your production environment
json_class:         Chef::Environment
name:               production
override_attributes:

Development/chef-repo »
```

图 8-11　确认环境创建

可以用 testqa 代替 knife environment 命令中的 production，对 testqa 环境做相同的操作。

8.6.3　为每个服务器分配一个环境

下一组步骤将温习更改节点所属环境的方法。环境可能包含独特的属性信息（例如，数据库 IP 地址）。在本练习中，我们将按照前一练习创建的两个环境配置每个虚拟机。

打开 Web 浏览器并转到 https://manage.chef.io。用本章开始时设置的凭据登录。

单击界面顶端的 Nodes，然后单击第一台注册服务器（亦称节点），屏幕下半部分将显示节点的详细信息。

1. 为该节点分配两个已创建环境（production 或 testqa）中的一个，如图 8-12 所示。

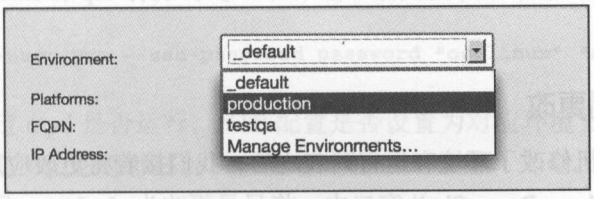


图 8-12　为节点分配已创建的环境

2. 选择之后，将确认环境更改。单击 Save 确认更改。

3. 单击用托管 Chef 账户注册的第二个节点，并为其分配其他环境（与第一个节点选择的相反）。

主节点界面将显示两个注册的节点和每个节点分配的环境。

8.6.4　修改每个服务器的运行列表，以运行 Managedroot 烹调书

打开浏览器转向 https://manage.chef.io。用本章开始时设置的凭据登录。

单击界面顶端的 Nodes，然后单击第一台注册服务器（亦称节点），屏幕下半部分将显示节点的详细信息。

1. 向下滚动到节点配置的 Run List 部分。如果该节点之前配置为运行 NTP 烹调书，那么就行了。

单击 Run List 部分的 Edit（编辑），将显示新屏幕，可以在其中修改运行列表，如图 8-13 所示。

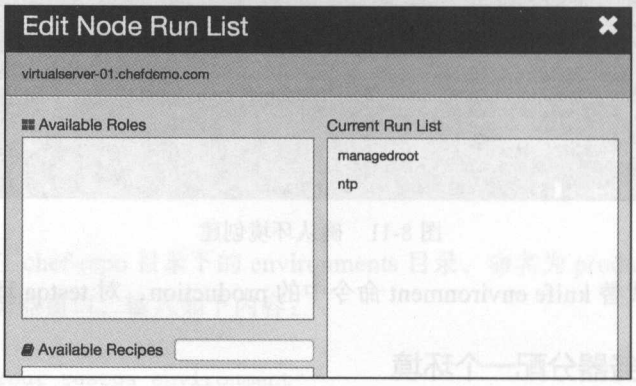


图 8-13 编辑节点运行列表


注意 记住，Chef 中顺序很重要。运行列表中先指定的菜单首先执行。

2. 对第一个节点单击 Save Run List（保存运行列表）。

3. 单击第二个注册服务器并重复第 1 步和第 2 步。确保 ntp 和 managedroot 食谱都在运行列表中。

8.6.5 对节点应用更改

为两个虚拟客户机修改了环境和运行列表之后，我们接着将更改应用到节点上。

在终端或者 Windows PowerShell 窗口中，将目录更改为 chef-repo 目录。

执行如下命令：

```
knife ssh --ssh-远程用户名--ssh-password 远程用户密码
"os:linux" "sudo chef-client"
```

例如：

```
knife ssh --ssh-user yvo --ssh-password password "os:linux" "sudo
chef-client"
```


注意 根据你的 sudo 配置，在执行上述命令时可能提示输入 sudo 密码。

输出将在窗口中滚动显示，通知你 Chef 客户端正在对两个虚拟客户机进行更改。完

成时，你将看到类似“ Chef client finished, X/10 resources updated”（ Chef 客户端执行结束，X/10 个资源更新）的输出。根据节点所分配的环境，更新的资源可能不同。

8.6.6　校验实施的策略

Chef 客户端可能已经在两台服务器上运行，而且更改已经实际生效。我们使用 knife ssh 确定每个机器上的一些配置文件。

在终端或者 Windows PowerShell 窗口中，将目录更改为 chef-repo 目录。

我们首先确定每个节点的 /etc/shadow 是否与通过策略（烹调书和环境）设置的预期状态相符，如图 8-14 所示。

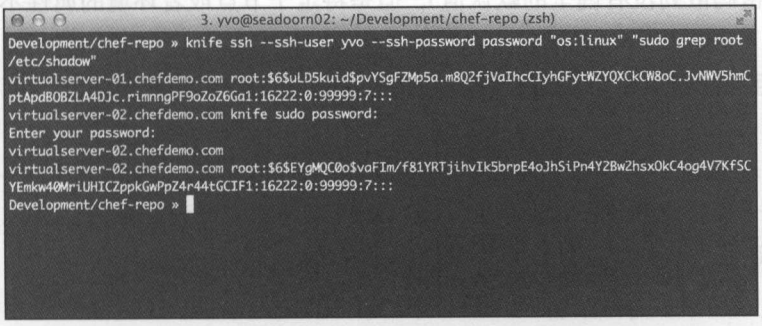


图 8-14　通过搜索校验 /etc/shadow 的更改

```
knife ssh --ssh-user 远程用户名  --ssh-password 远程用户密码
"os:linux" "sudo grep root /etc/shadow"
```

例如：

```
knife ssh --ssh-user yvo --ssh-password password "os:linux" "sudo grep root
/etc/shadow"
```

接下来，我们确定 ntpd 是否运行，NTP 配置是否设置为对应环境文件中描述的服务器。可以使用 knife ssh 确定（参见图 8-15）：

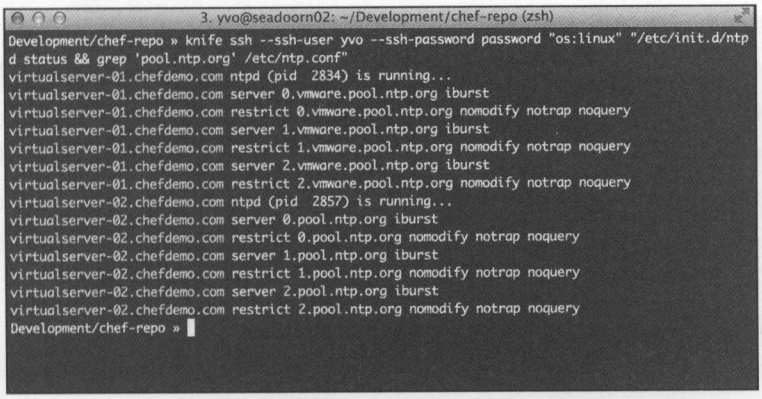


图 8-15　通过搜索校验 ntpd 的安装和启动

```
knife ssh --ssh-user 远程用户名  --ssh-password 远程用户密码
"os:linux" "/etc/init.d/ntpd status && grep 'pool.ntp.org' /etc/ntp.conf"
```

8.7 小结

本章以前一章为基础，很快地深入到托管 Chef 账户的建立。本章还介绍了社区烹调书，如何下载社区烹调书，还有更重要的——如何为两个虚拟客户机设置预期状态配置。

需要注意的是不修改烹调书代码的最佳实践。在本章中，作为一般做法，独特的属性应该在烹调书之外设置。每本烹调书可能自带一组“健全的”默认值（在 managedroot 烹调书的例子中，根密码散列没有健全的默认值），但是特定于节点或者环境的属性不应该在烹调书中直接建立。

在 Chef 组织的节点中可以进行各种系统管理任务，你可以接受挑战，自行尝试更多的 Chef 社区烹调书（例如，安装一个 SQL Server 实例，通过烹调书管理所有用户等）。

第 9 章的基础是引导、指定运行列表等功能，并将其与 Knife 插件 knife-vsphere 结合，你可以使用这个插件管理 vSphere 基础设施；还可以用它引导新服务器，为你所启动的服务器分配环境、运行列表，或者修改现有的虚拟客户机。

参考文献

[1] http://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=1006427

[2] Full URL: http://github.com/yvovandoorn/managedroot/zipball/master/

第 9 章 Chapter 9

用 Chef 管理 VMware vSphere

你需要一台安装了 ChefDK 的机器，以了解和实施本章中包含的主题。

在本章中，我们超越 Chef，找出 Chef 与各个 VMware 管理层次（如 ESXi 或者 vSphere vCenter）的不同整合点。Chef 提供了两个很好的 VMware 整合点：Knife 和 Chef 配给。

本章包含如下主题：

- Knife 插件
- Chef 配给

正如前几章所学习到的，Knife 是一个驻留于管理服务器 / 工作站之上的命令行工具。Knife 可以扩展，不仅仅能和 Chef Server 通信（不管 Chef Server 是在你的数据中心之内，还是像本书中一样，是由 Chef 维护的托管 Chef 服务）。Knife 的扩展能力是通过插件系统完成的，用户可以在该系统中发出 chef gem install knife-vsphere 或 chef gem install knife-esxi 命令。

Chef 配给是 Chef 所开发的相对新颖的技术。Chef 配给将食谱的编写提升到全新的水平。在第 8 章中，使用 Chef 编写了管理虚拟机（VM）的烹调书。Chef 配给背后的概念是，它使用相同的观念，但是仅仅应用到一台完整的机器。可以利用机器资源，创建描述完整应用程序栈的食谱。就像 Chef 在单一节点（或者机器）上运作时一样，在描述机器的食谱上，顺序也很重要。如果在编排整个应用程序栈时考虑需求 / 相互依赖性，确保在托管需要数据库服务器的应用服务器启动之前，该数据库（以及合适的内容）已经就绪，那么这一点就很重要。

前面几章利用了 ChefDK，本章还将继续利用该程序。本章有意地减少第 8 章中的实操环节，因为在此利用了第 8 章已经介绍的一些练习，以帮助读者理解与 knife-vsphere 或者 Chef 配给之间的联系。

下面是 knife-vsphere 支持的一些功能：

- 可以访问 https://manage.chef.io，运行 ChefDK 的工作站
- 可以利用 vSphere vCenter 服务器完成的如下工作
  - 根据基于 RHEL6 的映像克隆 VM
  - 创建自定义模板
- 我们在第 8 章中创建的 chef-repo 文件夹
- 可以访问 https://manage.chef.io 的浏览器（Firefox 20+、Google Chrome 20+ 或 IE 10+）
- Text、Notepad++ 或 Bluefish 等文本编辑器

9.1 Knife 插件

Knife 最大的优势之一是通过添加插件扩展自身的能力。Knife 利用 Ruby gems 为自身增加功能。有了 ChefDK，可以发出 chef gem install plugin 命令，使用 ChefDK 自带的 Ruby 解释程序。

管理 VMware 环境的两个最流行的 Knife 插件是 knife-esxi 和 knife-vsphere。knife-esxi 插件扩展 Knife，使用文档中现有的 VMware 应用程序编程接口（API）处理 VMware 和 Chef Server 调用，与环境中的单独 ESXi 虚拟化管理器通信。

knife-vsphere 插件设计用于扩展 Knife，与 vSphere vCenter 服务器及 Chef Server 通信。knife-vsphere 的常见用例之一是克隆新映像，并从命令行（不管是 Linux、Mac OS X 还是 Windows 的命令行）直接使用自定义规格。因为 Knife 直接通过 API 与 vSphere 通信，对 VMware 的 PowerCLI 没有任何依赖。

knife-vsphere 支持的功能包括：

- 列表
  - VM
  - 文件夹
  - 模板
  - 数据存储
  - VLAN（目前需要 vSphere 分布式交换机）
  - 资源池
  - 群集
  - 自定义规格
  - 池或者群集中的主机
- VM 运营
  - 开 / 关电源
  - 克隆（包含可选的 Chef 引导和运行列表）
  - 删除

- VMDK 添加
- 迁移
- 连接 / 断开网络

■ 特定于克隆的自定义选项（对于 Linux 客户机）

- 目标文件夹
- CPU 核心计数
- 内存大小
- DNS 设置
- 主机名 / 域名
- IP 地址 / 默认网关
- VLAN（目前需要分布式 vswitch）
- 资源池

9.1.1　knife-vsphere 入门

我们首先需要安装 knife 插件。随着 ChefDK 的发行，这一过程已经变得极其简单。在前面几章已经安装了 ChefDK，我们再次使用该程序安装 knife-vsphere 插件。

打开一个终端或者 Windows PowerShell 窗口，将目录改为 chef-repo 目录，简单的例子是在 OS X 上输入 cd ~/Development/chef-repo 或者在 Windows 上输入 cd$env:userprofile\Development\chef-repo。

现在，按照图 9-1 的演示，输入 chef gem install knife-vsphere。

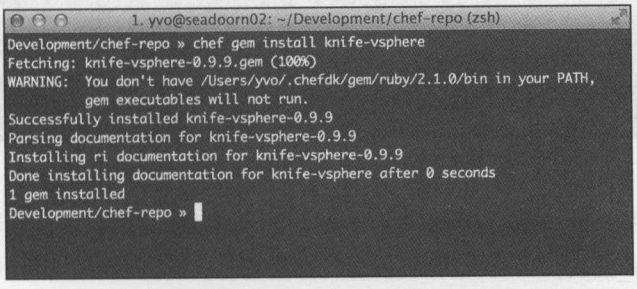


图 9-1　安装 knife-vsphere 插件

注意　由于 ChefDK 的更新，或者使用 Chef 和 ChefDK 进行本书之外的实验（例如，安装其他 Gems），可能会输出与屏幕截图不同的结果。

9.1.2　配置 knife.rb 文件

成功安装 knife-vsphere 之后，我们必须进行一些配置，使 Chef 与 vSphere 服务器通信。

在 chef-repo 目录中有一个 .chef 目录，包含用于正确签署 Chef Server API 请求的私钥文件，还包含 Knife 配置文件——knife.rb，如图 9-2 所示。

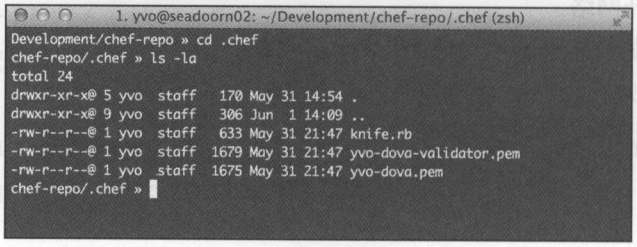


图 9-2 .chef 目录

使用可靠的文本编辑器（不是指 Windows 上的写字板 / 笔记本或者 OS X 上的 TextEdit）如 Notepad++、Sublime Editor 或 Bluefish，打开 .chef 目录中保存的 knife.rb 文件（参见图 9-3）。

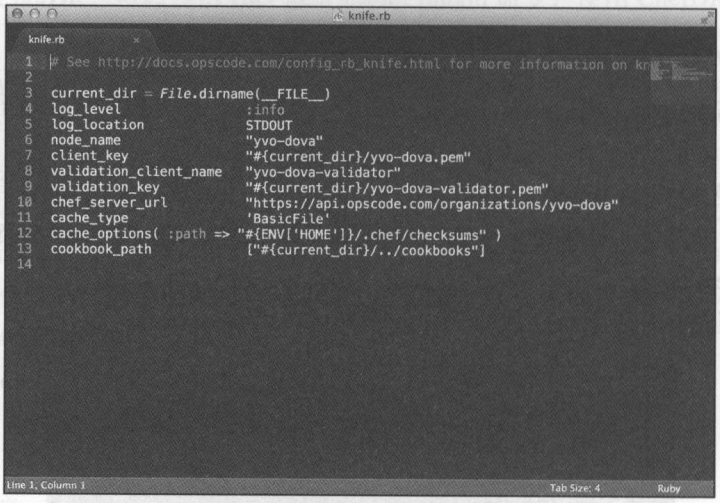


图 9-3 在文本编辑器中打开 knife.rb

添加如下代码行，包含 knife-vsphere 插件所使用的 vSphere 环境的相关信息：

```
knife[:vsphere_host] = "Your.Host.Or.IP.Address.Here"
knife[:vsphere_user] = "YourUserNameHere"
knife[:vsphere_pass] = "YourPasswordHere"
knife[:vsphere_dc] = "YourvSphereDCHere"
```

如果 vSphere 服务器使用自签署或者非标准证书管理机构的安全套接字层（SSL）证书，可以选择添加如下行：

```
knife[:vsphere_insecure] = true
```

完成之后，添加必要的行，在文本编辑器中保存文件并返回终端或者 Windows PowerShell 窗口。

9.1.3　校验配置

在终端或者 Windows PowerShell 中，执行 knife vsphere datastore list 命令校验是否可以成功地连接到 vSphere vCenter 服务器，如图 9-4 所示。

图 9-4　确认已连接到 vSphere vCenter 服务器

根据你的环境，上述命令将返回与 knife.rb 文件中指定的 vSphere DC 关联的可用数据存储。因为我们使用了 list 命令，对 vSphere 配置不会有实际的更改。

9.1.4　组合

如前所述，Knife 的真正力量在于结合 Chef Server 和 vSphere 服务器。在下面的例子中有一些假设。在第 8 章中，虚拟机的配给、引导和运行列表中添加项目等工作在不同的步骤中完成。

如果有一种方法将所有步骤串联成一个命令该多好：

1. 用特定的内存 /CPU/IP 设置配给主机。
2. 引导包含当前版本 Chef 的主机。
3. 在同一条命令中定义运行列表，使节点在安装 Chef 客户端之后立即执行。

遗憾的是，这是本练习中留给大家完成的部分，因为你的 vCenter 服务器可能以各种不同的方式设置。

所以，下面是一些假设：

- 你有为基于 RHE 的操作系统（包括 CentOS 或者 OEL）设计的自定义规格。
- 你有保存为 RHEL6 OS 类型的 VM 模板。
- 你有能力克隆模板。

注意　虽然 VMware 中可以选择 CentOS 或者 Scientific Linux 作为客户操作系统，但是为了正常使用自定义规格，模板必须保存为基于 Red Hat 的 OS。在我的测试中，除了 CentOS 模板被显示为 Red Hat 映像之外，没有任何害处。

下面是在一步之中完成以上所有工作的命令样板：

```
knife vsphere vm clone YourVMName --template YourLinuxTemplate
--cips YourVMIPaddress --cdnsips YourVMDNSServer --chostname
```

```
UnQualifiedDNSNameforCustomisation --cdomain DomainNameforCustomisation
--start --bootstrap true --distro chef-full --ssh-user SSHUser
--ssh-password SSHPassword --run-list "ARunList"
```

例如：

```
knife vsphere vm clone awebapp01 --template centbase --cips 172.31.8.101/22
--cdnsips 8.8.8.8 --chostname awebapp01 --cdomain opscode.us --start
--bootstrap true --distro chef-full --ssh-user username --ssh-password
Password --run-list "role[base],role[webserver]"
```

我们来分解上述命令所完成的具体工作：

- knife vsphere vm clone：这条命令告诉 Knife 和 vSphere，我们将要克隆一个映像。
- *YourVMName*：这是 vSphere 中显示的 VM 名称。
- --template *YourLinuxTemplate*：这是我们用于克隆的具体模板名称。
- --cips *YourVMIPaddress*：（可选，使用自定义模板）以无类域间路由（CIDR）标记法指定 IP 地址（例如，172.31.8.101/22）。
- --cdnsips *YourVMDNSServer*：（可选，使用自定义模板）为 VM 指定 DNS 服务器。
- --chostname *UnQualifiedDNSNameforCustomisation*：（可选，使用自定义模板且主机名没有在域名系统 [DNS] 中注册）指定主机名；如果你尚未注册 DNS，这被证明非常有用。
- --cdomain *DomainNameforCustomisation*：（可选，使用自定义模板）指定主机域名。
- --start：命令 VM 在克隆之后启动。
- --bootstrap true：命令 VM 在克隆和启动之后引导。
- --distro chef-full：命令 VM 使用默认引导模板。
- --ssh-user *SSHUser*：用于引导之后连接到 VM 的用户。
- --ssh-password *SSHPassword*：用于引导之后连接到 VM 的密码。
- --run-list "*role[base],role[webserver]*"：应该用于引导新的 VM 并保存到 Chef Server 作为定义运行列表的运行列表。

在执行这条命令之后，工作站将与 3 个单独的目标交互，使新机器上线，完成该过程需要图 9-5 中的 5 个不同阶段。

- 第 1 阶段：VM 创建

  Knife 将与 vSphere 通信，以克隆 VM。
- 第 2 阶段：自定义

  vCenter 服务器将对新 VM 应用自定义模板。
- 第 3 阶段：引导

  在 vCenter 完成之后，它将通知 Knife。Knife 将等待新主机上的安全外壳（SSH）可用。SSH 启动之后，Knife 将复制验证器密钥、样板 client.rb，并执行脚本下载 Chef。（默认情况下，设置为与 chef.io 通信；但是你可以将此更改为指向内部主机。）

- 第 4 阶段：注册

  Chef 客户端在主机上安装并自行注册到 Chef Server（使用验证器密钥）。新 VM 上的客户端还向服务器传达运行列表。

- 第 5 阶段：校验

  Chef 客户端第一次运行之后，Knife 将通过服务器校验该节点在线，然后退出。

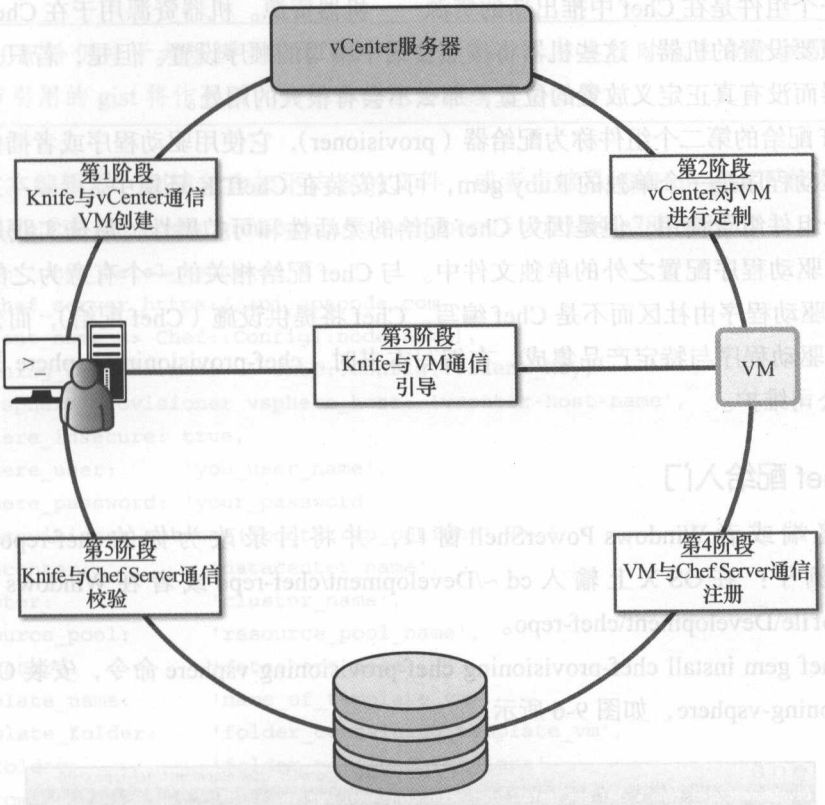


图 9-5　knife vsphere clone 的 5 个阶段

除了设置新 VM 之外，你还可以执行管理性任务，如列出可用的各种数据存储。knife-vsphere 输出的每条命令都是为了管理注册到 vSphere 的 VM。

9.2　Chef 配给

Chef 配给于 2014 年中期发布，将 Chef 提升了一个级别，已经超越了单个节点上单独配置和应用程序部署的管理。Chef 配给能够管理整个应用程序栈或者群集，其中编排是关键。Chef 配给通过代码完成这些任务，并对整个栈的配置应用相同的思想（例如，顺序很重要，可执行策略保存在食谱中）。

knife-vsphere 的重点在于单个 VM 和单个 VM 上的运行列表配置，而 Chef 配给可以管理 VMware 环境中的整个服务器池。

9.2.1 Chef 配给架构

Chef 配给有两个组件：

- 第一个组件是在 Chef 中推出新的资源——机器资源。机器资源用于在 Chef 食谱中声明想要设置的机器。这些机器将按照食谱中编写的顺序设置。但是，若只是定义一堆机器而没有真正定义放置的位置，那么不会有很大的用处。
- Chef 配给的第二个组件称为配给器（provisioner），它使用驱动程序或者插件方法。每个驱动程序是一个单独的 Ruby gem，可以安装在 ChefDK 环境中。

这两个组件组合使用，但是因为 Chef 配给的灵活性和可扩展性，最佳实践是将机器资源配置放在驱动程序配置之外的单独文件中。与 Chef 配给相关的一个有意为之的设计决策是，大部分驱动程序由社区而不是 Chef 编写。Chef 将提供设施（Chef 配给），而公司或者开源用户提供驱动程序与特定产品集成。在编写本书时，chef-provisioning-vsphere 驱动程序由 Rally 软件公司维护。

9.2.2 Chef 配给入门

打开终端或者 Windows PowerShell 窗口，并将目录改为你的 chef-repo 目录，举个简单的例子：在 OS X 上输入 cd ~/Development/chef-repo 或者在 Windows 上输入 cd $env:userprofile\Development\chef-repo。

键入 chef gem install chef-provisioning chef-provisioning-vsphere 命令，安装 Chef 配给和 chef-provisioning-vsphere，如图 9-6 所示。

```
1. yvo@seadoorn02: ~/Development/chef-repo (zsh)
Development/chef-repo » chef gem install chef-metal chef-metal-vsphere
Fetching: inifile-2.0.2.gem (100%)
WARNING:  You don't have /Users/yvo/.chefdk/gem/ruby/2.1.0/bin in your PATH,
          gem executables will not run.
Successfully installed inifile-2.0.2
Fetching: cheffish-0.6.2.gem (100%)
Successfully installed cheffish-0.6.2
Fetching: chef-metal-0.12.1.gem (100%)
Successfully installed chef-metal-0.12.1
Parsing documentation for chef-metal-0.12.1
Installing ri documentation for chef-metal-0.12.1
Parsing documentation for cheffish-0.6.2
Installing ri documentation for cheffish-0.6.2
Parsing documentation for inifile-2.0.2
Installing ri documentation for inifile-2.0.2
Done installing documentation for chef-metal, cheffish, inifile after 2 seconds
Fetching: chef-metal-vsphere-0.2.0.gem (100%)
Successfully installed chef-metal-vsphere-0.2.0
Parsing documentation for chef-metal-vsphere-0.2.0
Installing ri documentation for chef-metal-vsphere-0.2.0
Done installing documentation for chef-metal-vsphere after 0 seconds
4 gems installed
Development/chef-repo »
```

图 9-6 安装 chef 配给和 chef-provisioning-vsphere


注意　因为 ChefDK 发布以来的更新，或者你在本书范围之外试验 Chef 和 ChefDK（例如，安装其他 gem），输出可能与屏幕截图中不同。

9.2.3　启动某些节点


警告　Chef 配给仍处于大规模开发之中，很有可能在本书发行时有些命令做了小的修改。本节引用的 gist 将作更新，确保示例能够继续正常工作。

打开文本编辑器，创建包含如下内容的文件，或者直接从 GitHub gist 下载内容：http://bit.ly/1nv8GHn（或 https://gist.github.com/yvovandoorn/519f91d84e786fe4b5d7）。

```
require 'chef_metal_vsphere'
with_chef_server https://api.opscode.com,
  :client_name => Chef::Config[:node_name],
  :signing_key_filename => Chef::Config[:client_key]
with_vsphere_provisioner vsphere_host: 'vcenter-host-name',
  vsphere_insecure: true,
  vsphere_user:     'you_user_name',
  vsphere_password: 'your_password'
with_provisioner_options('bootstrap_options' => {
  datacenter:          'datacenter_name',
  cluster:             'cluster_name',
  resource_pool:       'resource_pool_name',
  datastore:           'datastore_name',
  template_name:       'name_of_template_vm',
  template_folder:     'folder_containing_template_vm',
  vm_folder:           'folder_to_clone_vms_into',
  customization_spec: 'standard-config',

  ssh: {
    user:                   'username_on_vm',
    password:               'name_of_your_first_pet',
    port:                   22,
    auth_methods:           ['password'],
    user_known_hosts_file: '/dev/null',
    paranoid:               false,
    keys:                   [ ],
    keys_only:              false
    }
  })
```

将文件保存到 chef-repo 目录下，命名为 vsphere-metal.rb。

注意，必须配置 vsphere-metal.rb 文件中的相关部分，需要配置的项目包括：

- vcenter-host-name
- vsphere_user
- vsphere_password
- datacenter
- cluster
- resource_pool
- datastore
- template_name
- template_folder
- vm_folder
- customization_spec
- user
- password

注意 为了使 Chef 配给正常工作，在新 VM 中引导的用户必须启用 NOPASSWD sudo 权限。

再打开一个文本编辑器窗口，创建有如下内容的文件，或者从 GitHub gist 直接下载内容：http://bit.ly/1nv8GHn（或 https://gist.github.com/yvovandoorn/519f91d84e786fe4b5d7）：

```
require 'chef_metal'

with_chef_server "https://api.opscode.com",
  :client_name => Chef::Config[:node_name],
  :signing_key_filename => Chef::Config[:client_key]

1.upto 2 do |n|
  machine "metal_#{n}" do
    action [:create]
    recipe 'ntp'
    converge true
  end
end
```

将两个文件都保存在 chef-repo 目录下。

打开终端或者 Windows PowerShell 窗口，并将目录改为你的 chef-repo 目录，例如，在 OS X 上输入 cd ~/Development/chef-repo 或者在 Windows 上输入 cd $env:userprofile\Development\chef-repo。

为了执行 Chef 配给，我们将直接在你的工作站（而非 VM）上执行 Chef 客户端。

执行如下命令：

```
'chef-client -z vsphere-metal.rb machine.rb'
```

这将启动驻留内存的 Chef Server，并用网络时间协议（NTP）运行列表配给两台主机。在配置中指定 use_chef_server，加入的节点将注册到你的托管 Chef 账户。

想象一下，不仅有关于如何在一台服务器上安装特定应用程序的社区烹调书，还有按照你所定义的顺序定义整个虚拟基础设施的完整烹调书，而且可以测试确保其有效。可以和基础设施剧本或者 wiki 的维护说再见了，这一切都可以在一个食谱中处理。

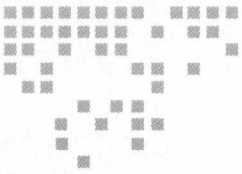
9.3　小结

本章介绍了使用 Chef 管理 vSphere 环境的不同方法。除了新主机的引导之外，Knife 还可以用于 vSPhere 环境的实际管理，Chef 配给可以用于创建包含特定应用或者整个基础设施的全栈配置的食谱，全面利用 vSPhere 环境。

第四部分 Part 4

Ansible

- 第 10 章　Ansible 简介
- 第 11 章　Ansible 系统管理任务



Chapter 10 第 10 章

Ansible 简介

Ansible 是 Michael DeHaan 开发的配置管理解决方案（也称为 Cobbler 部署系统）。*Ansible 的商业支持由 Ansible 公司提供。*

本章包含如下主题：

- Ansible 架构
- Ansible 组
- Ansible 临时命令执行
- Ansible 剧本
- Ansible 角色
- Ansible Galaxy

10.1 Ansible 架构

Ansible 用 Python 编写，是一个无代理的解决方案。安装了 Ansible 的服务器称作控制器，接受指令进行部署的服务器被称作托管节点。部署指令用 YAML 编写，控制器的指令用 Python 在托管节点上执行。控制器必须安装 Python 2.6 版本。托管节点必须运行 Python 2.4 或更高版本。由于 JSON 解释程序依赖性，建议在托管节点上运行 Python 2.5 或更高版本。

注意 Python 3 目前没有得到支持。如果你的 Linux 分发版本没有安装 Python 2.x，必须安装，最好是 2.5 或者更高版本，并在本章后面讨论的 Ansible 库存文件中设置 ansible_python_interpreter 变量。

控制器和托管节点之间的所有通信通过安全外壳（SSH）进行。因为 SSH 很大程度上是访问 Linux 服务器的标准，这简化了 Ansible 基础设施的准备：只要验证执行 Ansible 命令的用户在每个托管节点上的 authorized_keys 文件中有一个条目。如果系统管理员因为任何原因没有在 authorized_keys 文件中放入 SSH 密钥，也有替代的身份验证选项，如在 Ansible 执行期间使用 --user 和 --askpass 标志。对于需要提升权限的 Ansible 任务，你可以使用如下的运行时标志配对：

- --su 和 --ask-su-pass
- --sudo 和 --ask-sudo-pass（在托管节点的 sudoers 文件中必须有条目）

为了简单起见，第 10 章和第 11 章的代码示例假定你以控制器根用户的身份执行 Ansible，该用户的 SSH 密钥已经加入每个托管节点上的 authorized_keys 文件。

图 10-1 说明了控制器和托管节点之间的通信。

Ansible 和其他配置管理技术（如 Puppet 和 Chef）之间的显著差异之一是，控制器可以是任何机器，包括系统管理员的笔记本电脑。如果有一组管理员执行 Ansible 剧本，指定一台机器作为控制器可能是个好主意。Ansible 公司有一个处理这种情况的解决方案——Ansible Tower，它提供一个中心管理点，具备用户界面（UI）、应用程序编程接口（API）和命令行界面（CLI），供管理员团队跟踪剧本和运行的任务。本书不介绍 Ansible Tower，但是你应该了解它的管理用途。

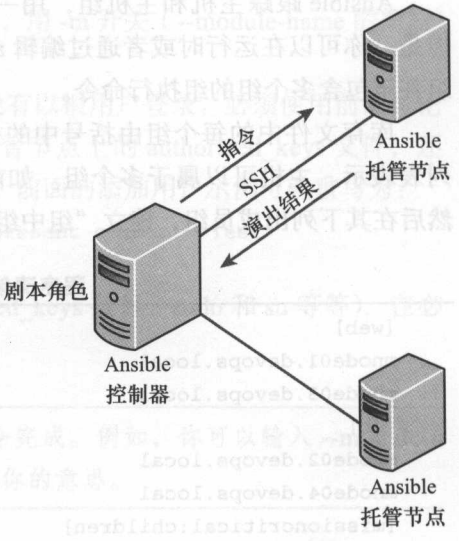


图 10-1　Ansible 组件拓扑

10.2　准备 Ansible 测试实验室

第 10 章和第 11 章使用 3 种机器设置：

- Ansible 控制器——Ubuntu 12.04
- 托管节点 1——Ubuntu 12.04
- 托管节点 2——CentOS 6.5

设置上述环境的最快速方法是使用 Vagrant（在第 3 章中讨论过）。资源分配为：控制器和托管节点 1——1vCPU 和 512MB RAM。我们最终将在托管节点 2 上安装数据库服务器，所以为其分配 2 ~ 4GB RAM。本书的 GitHub 页面有一个样板 Vagrantfile，你可以将其作为基础，构建测试实验室。

控制器默认操作系统软件包存储库中的 Ansible 版本可能已经过时，按照 Ansible 文档网

站上的说明，为控制器获取最新的稳定版本（http://docs.ansible.com/intro_installation.html）。

为了易用性考虑，一定要在两个托管节点上的 authorized_keys 文件中列出控制器节点的 SSH 公钥。否则，在托管节点上每次执行 Ansible 时都必须指定用户名和密码。

控制器需要包含为托管节点 IP 所创建记录的 DNS 服务器的访问权限，也可以在控制器的 /etc/hosts 文件中输入托管节点的条目。

10.3 Ansible 组

Ansible 跟踪主机和主机组，用一个库存文件进行管理，默认路径为 /etc/ansible/hosts。但是，你可以在运行时或者通过编辑 ansible.cfg 文件，指定该路径。可以对单独主机、主机组甚至包含多个组的组执行命令。

库存文件中的每个组由括号中的标签标识（例如，[group_name]）和组标签下该组主机列表表示。主机可以属于多个组。如前所述，也有可能在上级组标签的最后附加 :children，然后在其下列出成员组，建立“组中组”。程序清单 10-1 展示了示例。

程序清单 10-1 Ansible 库存文件

```
[web]
mnode01.devops.local
mnode03.devops.local
[db]
mnode02.devops.local
mnode04.devops.local
[missioncritical:children]
web
db
```

可以使用通配符、范围等创建复杂的库存文件（例如，mnode[01:10].devops.local）。如果想要验证库存文件有效性，可以使用 ansible all --list-hosts 命令。

10.4 Ansible 临时命令执行

Ansible 可以执行的命令称作模块。例子包括 ping、command、user 和 setup。可以在 Ansible 控制器上键入 ansible-doc -l 命令查看完整的模块列表。ansible-doc 命令还能提供每个模块的详细信息，格式类似于 Linux 的 man 页面（例如，ansible-doc yum）。用户可以自由地创建自己的模块，添加 Ansible 当前不具备的功能，也可以用控制器操作系统支持的任何语言编写这些模块。

对于大部分配置管理工作流，你将利用剧本执行模块，但是在少数情况下，临时命令

执行可能更方便：验证托管节点的可访问性、执行一次性任务（如系统重启）等。程序清单 10-2 展示了一些临时 Ansible 模块执行的例子。

程序清单 10-2　Ans ible 临时命令执行

```
ansible all -m ping
ansible mnode02.devops.local -m command -a '/sbin/reboot'
ansible web -m user -a 'name=vmtrooper state=present comment="Test" uid=1000'
ansible web -m user -a 'name=vmtrooper state=absent remove=yes'
```

临时命令以可执行程序名 ansible 开始，然后是模块将要执行的服务器模式，该模式包含明确的主机名或者库存文件中的一个组标签。接下来，用 -m 开关（--module-name 的简写）表示执行的模块。

临时命令将以当前用户的身份运行。所以，如果没有以根用户登录，必须使用前面讨论过的 --su 或 --sudo 选项。如果 SSH 密钥没有出现在托管节点上的 authorized_keys 文件，还有必要使用 --ask-pass 选项。为了让非根用户执行代码，前面的添加用户示例可就重写为：

```
ansible web -m user -a 'name=vmtrooper state=present comment="Test"
uid=1000' --ask-pass --sudo --ask-sudo-pass
```

当然，需要根据公司的安全惯例（例如，authorized_keys 条目、sudo 和 su 等等），在必要时更改命令选项。


注意　除了选项缩写之外，Anshible 还支持选项的命令完成。例如，你可以输入 --module-n ping 代替 -module-name ping，Ansible 能够理解你的意思。

10.4.1　Ping 模块

Ping 模块验证执行命令的用户可以访问目标服务器。在程序清单 10-2 中，我使用 ping 命令加上 all 模式，验证 Ansible 库存中的所有服务器都可以供 Ansible 控制器访问。

10.4.2　Command 模块

Command 模块可以在目标服务器上运行本地可执行程序。必须用 -a（argument 的缩写）开关指定目标服务器上所要运行的可执行文件的完整路径。

10.4.3　User 模块

User 模块用于根据可能在多台服务器上部署的应用程序需求，添加或者删除用户。用参数开关可以指定的数据包括用户名、用户存在或者缺失、UID 和关于用户的其他信息。你可能已经注意到，每个模块的参数结构可能不同。一定要参考 ansible-doc 命令，验证使用模块的正确语法。

10.4.4 Setup 模块

在我们继续之前，先看看一个较为关键的模块：setup 模块。这个模块帮助我们收集库存文件中各系统的相关事实（例如，操作系统、虚拟化管理器等）。和其他配置管理系统一样，这些系统事实可用于做出关于命令在目标服务器上如何执行的重要决策。此时，我建议运行 setup 模块（ansible all -m setup）了解关于托管节点的系统事实。当我们讨论多平台上的 Ansible 剧本执行时，将更详细地研究系统事实是如何帮助我们优化代码的。

10.5 Ansible 剧本

剧本（playbook）是部署指令的集合，这些指令也称为演出（plays）。使用剧本的好处是可以组合多个 Ansible 模块，执行应用程序试运行。而且，剧本的强制性执行规定了 Ansible 指令的执行顺序，使其他系统管理员更容易理解逻辑流程。剧本应该保存在 Git 或者其他源代码管理（SCM）系统中，便于配置管理工作流的维护。

Ansible 剧本用 YAML 编写，在第 5 章中已经学习了有关 YAML 的知识，所以语法看起来有些熟悉。我们先来看一个简单的剧本，该剧本在一个托管节点上安装 Web 服务器（参见程序清单 10-3）。

程序清单 10-3 Web 服务器剧本：apache.yml

```
---
- hosts: web
  tasks:
    - name: deploy apache
      yum: pkg=httpd state=latest
    - name: apache config file
      copy: src='files/httpd.conf' dest='/etc/httpd/conf/httpd.conf'
      notify:
      - restart httpd
    - name: apache service runs
      service: name=httpd state=started
  handlers:
      - name: restart httpd
        service: name=httpd state=restarted
```

apache.yml 保存在 apache 目录下，该目录有两个子目录：files 和 templates。剧本的第一行使用 hosts: 关键字指定运行这些指令的主机（/etc/ansible/hosts 文件中的 servers 组）。Web 主机组和属于它的所有任务组成了一次演出。

注意 一个剧本中可以有多场演出。如果我们想在另一个主机组（例如，db 组）上执行任务，hosts: db 声明及其任务被视为剧本中的另一场演出。

接下来，tasks: 关键字表示 Web 组中主机上将要执行的模块的开始。每个任务至少包括两个部分，name: 是一个标识符，让阅读代码的任何人知道这个特定任务的作用。我们将会在下面的代码中看到这个名称值被引用。任务的下一个部分是要执行的 Ansible 模块——对于我们的第一个任务，是 yum: 模块。我们只关心两个参数：pkg 告诉 yum 部署 httpd 软件包，state 指定应该安装 httpd 的最新版本。如果不关心软件包的版本，也可以指定 present。

在第一个任务中，你应该注意到，我们明确地调用 CentOS 的 yum 软件包管理器。Ansible 没有通用的软件包安装程序模块。这使得剧本作者可以指定所要使用的特定包管理器。在后面的例子中，你会看到使用条件语句根据运行的操作系统进行不同的操作（也就是说，对于基于 Debian 的系统使用 apt，对于基于 Red Hat 的系统使用 yum）。

下一个任务更有趣，因为我们使用了 notify: 参数。这个任务恰当地命名为 apache config file，这是因为我们用其定义 httpd 使用的配置文件。在这个任务中，我们使用 copy: 模块和两个参数：src 是我们要复制的文件与剧本文件的相对位置；dest 是目标服务器上文件所要复制的位置。notify: 任务参数表示响应配置文件更改时执行某些功能的 Ansible 处理程序名称。

最后一个任务告诉 Ansible 确保软件包部署之后运行 httpd 服务。这一任务使用 service: 模块和两个参数：name 指定所要管理的服务；states 指定服务应该运行还是停止。

演出中的 handlers: 部分和 tasks: 部分很类似。但是，这里列出的项目起着特殊的作用——响应通知操作。我们在这里用 name: 行定义 restart httpd 处理程序，以及我们用于启动 httpd 服务的 Ansible 模块（service:）。需要了解的重要处理程序行为之一是，在 Ansible 运行中，该处理程序仅执行一次。所以，如果我们对配置文件进行多次更改，该服务只在所有更改完成之后才重启。

保存和关闭该文件之后，可以在命令行使用 ansible-playbook 二进制程序执行它：

```
root@mnode01:~/apache# ansible-playbook apache.yml
```

在当前状态下，剧本必须从 Apache 文件夹中执行。以后，我们将研究如何使用 Ansible 角色对主机执行剧本，而无须担心路径。

条件表达式和变量

我们的 apache 剧本已经做好准备，为基于 Red Hat 的系统部署 Web 服务器。但是，如果我们的生产环境还有基于 Debian 的系统（例如，Ubuntu）怎么办？我们需要优雅地处理多种操作系统。

when: 条件语句允许根据偏好调整主机上执行的 Ansible 任务。在我们的例子中，将根据托管节点所属的操作系统家族，决定部署的软件包、使用的配置文件和管理的服务。程序清单 10-4 展示了经过修改的剧本，包含操作系统属于 Red Hat 和 Debian 家族的托管节点所用的指令。

程序清单 10-4 基于 Fact 变量执行模块的 Ansible when: 条件语句

```
---
- hosts: web
  tasks:
# Deploy the Web Server
  - name: Deploy Apache for Red Hat Systems
    yum: pkg=httpd state=latest
    when: ansible_os_family == 'RedHat'
  - name: Deploy Apache for Debian Systems
    apt: pkg=apache2 state=latest
    when: ansible_os_family == 'Debian'

# Copy the correct Config File
  - name: Apache Config File for Red Hat Systems
    copy: src='files/httpd.conf' dest='/etc/httpd/conf/httpd.conf'
    notify:
    - restart httpd redhat
    when: ansible_os_family == 'RedHat'
  - name: Apache Config File for Debian Systems
    copy: src='files/apache2.conf' dest='/etc/apache2/apache2.conf'
    notify:
    - restart httpd debian
    when: ansible_os_family == 'Debian'

# Generate the correct Web Content
  - name: Apache Content File for Red Hat Systems
    template: src='files/index.html' dest='/var/www/html/index.html'
      mode=0644
    when: ansible_os_family == 'RedHat'
  - name: Apache Content File for Debian Systems
    template: src='files/index.html' dest='/var/www/index.html' mode=0644
    when: ansible_os_family == 'Debian'

# Verify Web Service is running
  - name: Apache Service Runs for Red Hat Systems
    service: name=httpd state=started
    when: ansible_os_family == 'RedHat'
  - name: Apache Service Runs for Debian Systems
    service: name=apache2 state=started
    when: ansible_os_family == 'Debian'

# Restart Web Service in response to config file change
  handlers:
    - name: restart httpd redhat
      service: name=httpd state=restarted
```

```
  - name: restart httpd debian
    service: name=apache2 state=restarted
```

代码中已经添加了注释（前缀为 #），以澄清和帮助人们理解该剧本。Web 服务器部署中的每个重要步骤有两组指令：一组用于 Red Hat 系统，另一组用于 Debian 系统。Ansible 将求取 when: 语句的值，决定在目标节点上执行的任务，该语句使用 ansible_os_family 系统 fact 变量提供模块执行指导。

作为替代，我们可以使用 Ansible 的 group_by 函数，该函数能够自动根据某些条件（例如，ansible_os_family）创建更多的主机组。程序清单 10-5 展示了用 group_by 构造剧本的另一种方法。从来自 /etc/ansible/hosts 文件的 Web 主机组开始，告诉 Ansible 从 Web 主机组中创建附加组，以 ansible_os_family 系统事实作为选择条件。对于我们的系统，这意味着在运行时将创建两个组（一个用于 RedHat，另一个用于 Debian），这些主机组将用于剧本的其余部分。程序清单 10-5 展示了用于 RedHat 和 Debian 主机组的单独指令，它们不依赖于 when：条件语句。在结构中不包含条件语句是安全的，因为指令仅根据主机操作系统运行。例如，我们的 Debian 指令在 Red Hat 服务器上不会执行。

程序清单 10-5　根据事实执行模块的 Ansible `group_by` 函数

```
---
- hosts: web
  tasks:
  - name: Group Servers By Operating System Family
    action: group_by key={{ ansible_os_family }}
- hosts: RedHat
  tasks:
  - name: Deploy Apache
    yum: pkg=httpd state=latest
  - name: Apache Config File
    copy: src=files/httpd.conf dest=/etc/httpd/conf/httpd.conf
    notify:
    - restart httpd
  - name: Apache Service Runs
    service: name=httpd state=started
  handlers:
    - name: restart httpd
      service: name=httpd state=restarted

- hosts: Debian
  tasks:
  - name: Deploy Apache
    apt: pkg=apache2 state=latest
  - name: Apache Config File
    copy: src=files/apache2.conf dest=/etc/apache2/apache2.conf
```

```
    copy: src=files/apache2.conf dest=/etc/apache2/apache2.conf
    notify:
    - restart httpd
  - name: Apache Service Runs
    service: name=apache2 state=started
  handlers:
    - name: restart httpd
      service: name=apache2 state=restarted
```

程序清单 10-4 和 10-5 中的两个例子都解决了根据操作系统类型修改剧本执行的问题。但是，有许多重复的代码，Ansible 有额外的能力，可以帮助我们优化剧本。程序清单 10-6 展示了使用变量降低剧本中所需任务数量的方法。

程序清单 10-6 根据事实执行模块的 Ansible when：条件语句

```
---
- hosts: web
  vars_files:
"vars/{{ ansible_os_family }}.yml"
  tasks:
  - name: Deploy Apache for Red Hat Systems

    yum: pkg=httpd state=latest
    when: ansible_os_family == 'RedHat'
  - name: Deploy Apache for Debian Systems
    apt: pkg=apache2 state=latest
    when: ansible_os_family == 'Debian'
  - name: Apache Config File
    copy: src=files/{{ conffile }} dest={{ confpath }}
    notify:
    - restart httpd
  - name: Apache Service Runs
    service: name={{ webserver }} state=started
  handlers:
    - name: restart httpd
      service: name={{ webserver }} state=restarted
```

我们的剧本中有一个新元素：vars_files。这个关键字告诉 Ansible 我们在哪里保存剧本中使用的变量。程序清单 10-7 展示了包含变量值的 YAML 文件内容。

程序清单 10-7 RedHat.yml 和 Debian.yml 变量文件

```
#RedHat.yml
---
webserver: 'httpd'
conffile: 'httpd.conf'
```

```
confpath: '/etc/httpd/conf/httpd.conf'
contentpath: '/var/www/html/index.html'

#Debian.yml
---
webserver: 'apache2'
conffile: 'apache2.conf'
confpath: '/etc/apache2/apache2.conf'
contentpath: '/var/www/index.html'
```

在我们的剧本中，变量保存在剧本相同目录中的 vars 目录下。两个文件（RedHat.yml 和 Debian.yml）的名称对应于 ansible_os_family 系统事实变量返回的值。当我们想要使用任何变量（不管是系统事实还是用户定义变量）时，必须将其包含在双花括号中，使 Ansible 知道用存储的值代替其中的文本（例如，{{ ansible_os_family }}）。

因为 Ansible 要求明确指出托管节点操作系统的包管理器，我们仍然需要使用单独的任务，在基于 Red Hat 和基于 Debian 的系统上部署 Apache。但是，任务的其余部分使用通用模块（copy、service 等），这使我们可以使用导入的变量，消除重复任务。

Apache Config File 文件任务首先使用变量。copy 模块的 src 参数使用 conffile 变量，dest 参数使用 confpath 变量。即使处理程序定义也可以使用变量，我们看到，它使用 webserver 变量表示配置文件更改时将重启的服务。

如果比较程序清单 10-4 和 10-6 中的剧本，可以看出优化的代码需要的任务数少 3 个，处理程序少 1 个。这种节约看上去似乎不太明显，但是在更大的剧本中，使用变量可能导致更显著的代码优化。

10.6　Ansible 角色

目前，组织文件是个简单的任务，因为我们只处理 Apache 的部署，但是，如果我们想要部署其他类型服务器呢？我们要不断地在当前 YAML 文件中添加用于数据库、应用程序服务等的附加演出吗？即使应用模块化设计，将演出分隔到单独的 YAML 文件中，组织支持文件（如配置文件、变量文件等）的最佳方法又是什么呢？是将它们放在公共的目录中，还是根据功能创建不同的子目录？

Ansible 的作者考虑了这些问题，实现角色功能来处理它们。系统管理员可以根据相关文件实现的功能，用特定的目录结构分组它们。图 10-2 展示了根据 Apache Web 服务器角色建立的树形结构。

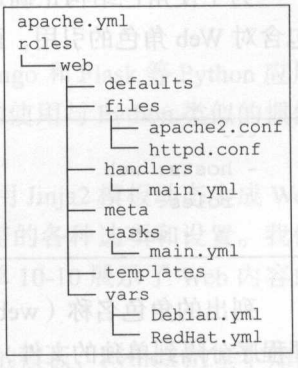


图 10-2　Ansible 角色目录结构

roles 目录和 apache.yml 文件在文件系统的同一级（例如，/src/apache.yml 和 /src/roles/），其余目录和文件在 roles 目录下（src/roles/web/files/、/src/roles/web/handlers 等）。你也可以将 Web 角色保存在 /etc/ansible/roles，我们在第 11 章中将采取这种做法。不过，为了简单起见，我们将把 Web 角色保存在 apache.yml 文件所在的目录。

在讨论适应角色结构所需的代码更改之前，我们先研究一下角色中每个文件夹的用途：

- defaults：如果角色在调用时接受参数（第 11 章中将看到一个示例），可以将包含未在主剧本角色中指定的默认值（例如，默认密码）的 YAML 文件放在这个目录下。
- files：打算复制到目标主机的任何文件。对于 Apache 服务器，可能是配置文件。
- handlers：执行操作响应剧本任务通知的指令（例如，在配置文件更改之后重启 Apache 服务器）。
- meta：当前角色所依赖的其他 Ansible 角色引用。
- tasks：剧本任务位置。
- templates：可用于根据系统事实或者其他变量动态生成内容的 Jinja2 文件。
- vars：剧本使用的变量。

注意 没有必要人工创建角色目录结构。Ansible 包含一个实用工具，为你生成该目录：ansible-galaxy init web。

Ansible Galaxy 在本章后面介绍，目前，知道该工具可以帮助你生成正确的角色结构就足够了，如果愿意，可以与 Ansible 社区分享。

利用角色目录结构，在访问各种目录中的内容时不用提供明确的路径。Ansible 可以根据使用的模块解释出所要查看的文件夹。例如，在程序清单 10-3 中，如果没有将配置文件保存在剧本文件的相同目录下，copy 模块的 src 参数要求明确的路径（copy: src=files/httpd.conf），当我们采用角色结构时，遵循正确的 Ansible 惯例，将配置文件放在角色的 files 子目录。在剧本中，只需指定文件名，Ansible 将自动解释文件的正确路径。

为了使角色结构正确运作，对剧本文件需要进行一些更改。首先，apache.yml 文件现在包含对 Web 角色的引用，而不是对实际演出任务和处理程序的引用（参见程序清单 10-8）。

程序清单 10-8 更新后的 apache.yml

```
---
- hosts: web
  roles:
     - web
```

列出的角色名称（web）必须与 roles 文件夹中的子目录名相符。然后，剧本的任务和处理程序分隔到单独的文件，这些文件都名为 main.yml，分别在 tasks 和 handlers 子目录下（参见程序清单 10-9）。

程序清单 10-9　保存在 Web 角色的任务（Tasks）和处理程序（Handlers）子目录下的 Apache 部署任务

```
#web/tasks/main.yml
---
include_vars: "{{ ansible_os_family }}.yml"
  - name: Deploy Apache for Red Hat Systems
    yum: pkg=httpd state=latest
    when: ansible_os_family == 'RedHat'
  - name: Deploy Apache for Debian Systems
    apt: pkg=apache2 state=latest
    when: ansible_os_family == 'Debian'
  - name: Apache Config File
    copy: src={{ conffile }} dest={{ confpath }}
    notify:
    - restart httpd
  - name: Apache Service Runs
    service: name={{ webserver }} state=started

#web/handlers/main.yml
---
include_vars: "{{ ansible_os_family }}.yml"
  - name: restart httpd
    service: name={{ webserver }} state=restarted
```

文件名 main.yml 指示 Ansible，这些任务和处理程序应该在另一个文件中引用角色时自动加载。角色结构需要以不同方式引用变量，我们使用 include_vars: 关键字代替 vars_files: 关键字，而且，不再需要指定 vars 子目录，因为 Ansible 将自动知道在哪里搜索变量文件。

这些更改初看似乎有些混乱，但是它们对根据代码完成的功能组织有很大的帮助。例如，如果采用 LAMP 栈工作，且有独立的 Web、数据库和 PHP 角色，这样做很方便。你可以对数据库角色进行更改，而不会影响 Web 和应用程序服务器角色。

模板

Ansible 支持 Jinja2 模板，用于生成角色内容。Jinja2 是 Django 和 Flask 等 Python 应用程序框架用于在运行时生成动态 Web 内容的技术，除了代码结构使用与 Python 类似的惯例之外，它类似于第 4 章中介绍的 Puppet 嵌入式 RuBy（ERB）技术。

Jinja2 模板对于配置文件和应用程序内容很有用，我们将使用 Jinja2 模板动态生成 Web 服务器的 index.html 文件，而不讨论生成 Apache 配置文件所必需的各种选项和设置。我们的 index.html 文件内容将根据目标主机的内容而变化。程序清单 10-10 展示了 Web 内容的 Jinja2 模板样板。

Jinja2 利用了许多 Python 惯例。所以，如果你想要开发自己的模板，Python 的基本知识很有益处。在我们的例子中，使用 Python 的字符串惯例求取文本，用一条 if-then-else 语句

确定打印输出的文本。

程序清单 10-10 index.html.j2 Web 内容模板

```
#jinja2: variable_start_string: '[%' , variable_end_string: '%]'
<html><body><h1>Ansible Rocks!</h1>
<p>This is the default web page for
{# Print the correct Operating System name with the right article i.e. "a"
vs "an"#}
{% if ansible_distribution[0].lower() in "aeiou" %}
  an
{% else %}
  a
{% endif %}
<b> [% ansible_distribution %] </b> server.</p>
<p>The web server software is running and content has been added by your
<b>Ansible</b> code.</p>
</body></html>
```

第一行是特殊指令，为 Jinja2 模板解释程序指明我们将使用非标准方法插入系统事实。这样做是因为标准的双花括号惯例有时会在 Ansible 中的模板上造成问题。

注意，Jinja2 语句块有不同的前缀：

- {# #} 包含注释。
- {% %} 包含条件逻辑和其他 Python 代码。我们的 if-then-else 语句决定在操作系统名称之前，哪一个冠词（a 或者 an）在语法上是正确的。
- [% %] 根据模板文件第一行的指令包含系统事实。同样，这一惯例可以根据模板第一行定制。在我们的例子中，使用 ansible_distribution 系统事实，检测目标服务器运行的操作系统。

有经验的 Python 开发者可能注意到，我们使用 endif 语句，这不是标准 Python 常见的做法，但是，对 Jinja2 理解条件语句的结束位置有帮助。

完成 Jinja2 文件后，我们将其放入 Web 角色的 templates 子目录，并在角色中创建一个任务以利用该模板（参见程序清单 10-11）。

程序清单 10-11 添加一个任务创建 HTML 内容

```
tasks/main.yml
---
include_vars: "{{ ansible_os_family }}.yml"
  - name: Deploy Apache for Red Hat Systems
    yum: pkg=httpd state=latest
```

```
  when: ansible_os_family == 'RedHat'
- name: Deploy Apache for Debian Systems
  apt: pkg=apache2 state=latest
  when: ansible_os_family == 'Debian'
- name: Apache Config File
  copy: src={{ conffile }} dest={{ confpath }}
  notify:
  - restart httpd
- name: Apache Content File
  template: src=index.html.j2 dest={{ contentpath }} mode=0644
- name: Apache Service Runs
  service: name={{ webserver }} state=started
```

我们添加名为 Apache Content File 的任务，使用 template 模块，复制模板并将其放入指定的目标路径。

10.7　Ansible Galaxy

Ansible Galaxy 是一个 Ansible 角色在线存储库，位置在 https://galaxy.ansible.com/。在开始为环境中的某些功能编写出色的 Ansible 角色之前，值得看看 Ansible 社区是否已经解决了该问题，并且张贴了可供使用的代码。

当你找到想要使用的 Ansible 角色时，用 ansible-galaxy install 命令指定名称（例如，ansible-galaxy install bennojoy.mysql）。然后，这个角色被放入 /etc/ansible/roles，立刻可以在任何剧本中使用。

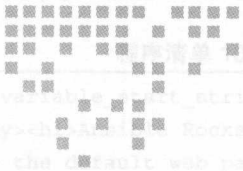
为 Ansible Galaxy 贡献代码相当简单。在网站的 Add a Role 部分，指定你的 GitHub 账户、包含角色的存储库以及角色在 Galaxy 网站上的标题（可选）。默认情况下，Galaxy 显示的角色名称为你的 GitHub 存储库名称。

10.8　小结

本章中学习了创建 Ansible 剧本部署 Web 服务器的基本知识。Ansible 角色可以帮助你组织剧本代码，使其可重用、可移植和易于维护。第 11 章介绍如何用目前为止发展的 Ansible 技能部署整个 LAMP 栈。

参考文献

[1] Ansible documentation. http://docs.ansible.com/



Chapter 11 第 11 章

Ansible 系统管理任务

本章的重点是使用 Ansible 进行多层部署，具体地说，本章介绍用 Ansible 部署 LAMP（Linux-Apache-MySQL-PHP）栈的一个例子。

本章包含如下主题：

- Web 服务器部署
- 应用层
- 数据库层
- 角色结构优化
- VMware 资源管理

11.1 Web 服务器部署

我们的 Web 角色所需的工作不多，因为在第 10 章中已经出色地完成了。但是，为了更好地组织代码，我们将把包含 Web 角色的文件夹移入 /etc/ansible/roles。

注意 下一步，我们将把所有 Ansible 角色放在控制器上的 /etc/ansible/roles 目录中。这允许我们从控制器上的任何位置执行 Ansible 剧本。角色路径可以通过编辑控制器上的 /etc/ansible/ansible.cfg 更新。

我们将要进行的另一个更改是添加一个任务，部署和启用网络时间协议（NTP），使服务器始终保持一致。在角色中添加的附加代码参见程序清单 11-1。

程序清单 11-1　在 /etc/ansible/roles/web/tasks/main.yml 中添加 NTP 部署任务

```
---
  - include_vars: "{{ ansible_os_family }}.yml"
  - name: Deploy Apache for Red Hat Systems
    yum: pkg=httpd state=latest
    when: ansible_os_family == 'RedHat'
  - name: Deploy Apache for Debian Systems
    apt: pkg=apache2 state=latest
    when: ansible_os_family == 'Debian'
  - name: Apache Config File
    copy: src={{ conffile }} dest={{ confpath }}
    notify:
    - config file update
  - name: Apache Service Runs
    service: name={{ webserver }} state=started
  - name: Deploy NTP on Ubuntu Servers
    apt: pkg=ntp state=latest
    when: ansible_os_family == 'Debian'
  - name: Deploy NTP for Red Hat Systems
    yum: pkg=ntp state=latest
    when: ansible_os_family == 'RedHat'
  - name: NTP Service Runs
    service: name={{ ntpserver }} state=started
```

记住，当你用 Ansible 添加新软件包时，需要提供一个任务，验证服务运行（如果那是预期的状态）。

11.2　应用层

在我们的部署中，应用层是一个简单的 PHP 服务器。PHP 安装在运行 Apache Web 服务器的同一台机器上。但是，我们仍然建立单独的 PHP 角色，而不是在现有的 Web 角色中添加新代码。如果我们以后想要切换不同类型的 Web 服务器或者应用服务器，这能够带来更大的灵活性。

程序清单 11-2 展示了用于部署 PHP 的剧本。

程序清单 11-2　/etc/ansible/roles/php/tasks/main.yml 中的 PHP 部署任务

```
---
  - include_vars: "{{ ansible_os_family }}.yml"
  - name: Deploy PHP packages for Red Hat Systems
    yum: pkg={{ item }} state=latest
    with_items: phppkg
    when: ansible_os_family == 'RedHat'
```

```
- name: Deploy PHP packages for Debian Systems
  apt: pkg={{ item }} state=latest
  with_items: phppkg
  when: ansible_os_family == 'Debian'

- name: Restart Apache after deploying PHP
  service: name={{ webserver }} state=restarted

- name: PHP Content File
  copy: src=index.php dest={{ phppath }}
```

这些任务看上去和我们的 Web 服务器部署任务类似，但是有一处显著差异。我们使用 with_items 循环机制，在一个软件包部署任务中安装多个软件包。with_items 循环接受一个 YAML 序列（类似于 Python 列表或者数组）或者包含 YAML 序列的变量名作为输入。（变量值参见程序清单 11-3。）剧本中另一个引人注目的任务是 Apache 服务的启动。有些发行版本可能需要这一步骤，使 Apache 知道 PHP 已经安装。

程序清单 11-3 PHP 角色变量文件

```
#/etc/ansible/roles/php/vars/RedHat.yml
---
webserver: 'httpd'
phppkg:
  - php
  - php-mysql
phppath: '/var/www/html/index.php'

#/etc/ansible/roles/php/vars/Debian.yml
---
webserver: 'apache2'
phppkg:
  - php5
  - php5-mysql
phppath: '/var/www/index.php'
```

11.3 数据库层

我们来看看如何用 Ansible 部署 MySQL。有趣的是，Ansible 自带管理 MySQL 和 PostgreSQL 数据库的模块。我们将在角色中利用这些模块，创建一个样板数据库和具有该数据库权限的用户。

我们的角色结构非常类似于 Web 和 PHP 角色中使用过的结构（参见程序清单 11-4）。

程序清单 11-4　/etc/ansible/roles/db/tasks/main.yml 中的 MySQL 部署任务

```
---
  - include_vars: "{{ ansible_os_family }}.yml"

  - name: Deploy MySQL for Red Hat Systems
    yum: pkg={{ item }} state=latest
    with_items: dbpkg
    when: ansible_os_family == 'RedHat'

  - name: Deploy MySQL for Debian Systems
    apt: pkg={{ item }} state=latest
    with_items: dbpkg
    when: ansible_os_family == 'Debian'

  - name: MySQL Service Runs
    service: name={{ dbservice }} state=started

  - name: Configure MySQL root password
    mysql_user: name=root password={{ dbpasswd }}
    ignore_errors: yes

  - name: MySQL DB Create
    mysql_db: name=test state=present login_user=root login_password=
      {{ dbpasswd }}

  - name: Set User Permission
    mysql_user: name=ansible password=ansiblerocks priv=test.*:ALL,GRANT
state=present login_user=root login_password={{ dbpasswd }}
```

“Configure MySQL root password”任务中的 ignore_errors 参数在多次运行剧本时抑制错误。这是可选的参数，但是如果没有，Ansible 将生成错误。

我们再次使用 with_items 循环，指示为 MySQL 安装的软件包列表。为了使 Ansible 的 MySQL 模块正常工作，需要安装 Python MySQL 库。所以，我们将其添加到两个变量文件中的软件包列表（参见程序清单 11-5）。

程序清单 11-5　db 角色变量文件

```
#/etc/ansible/roles/db/vars/RedHat.yml
---
dbpkg:
  - mysql-server
  - MySQL-python
dbservice: 'mysqld'
```

```
#/etc/ansible/roles/db/vars/Debian.yml
---
dbpkg:
  - mysql-server
  - python-mysqldb
dbservice: 'mysql'
```

用于密码参数的 dbpasswd 变量（参见程序清单 11-4）将由来自 defaults 目录的值或者剧本文件中 db 角色的参数化调用来设置。程序清单 11-6 展示了指定角色时使用参数的例子，程序清单 11-7 展示了保存在角色 defaults 目录中的默认数据库密码示例。

程序清单 11-6 从数据库剧本文件中进行参数化角色调用

```
#Playbook YAML file in your working directory that references the db role
# ~/db.yml
---
- hosts: db
  roles:
   - { role: db, dbpasswd: 'ansibleabc'}
```

程序清单 11-7 db 角色默认值文件

```
#/etc/ansible/roles/db/defaults/main.yml
---
dbpasswd: 'ansiblerocksabc'
```

具有安全意识的系统管理员会对默认密码以普通文本保存感到担忧。在 Ansible 版本 1.5 中，可以使用 ansible-vault，以 AES 算法加密角色中的任何文件，语法相当简单。创建 YAML 文件之后，使用 ansible-vault encrypt 命令加强其安全性。例如，如果数据文件名为 main.yml，语法为 ansible-vault encrypt main.yml。ansible-vault 命令询问密码，然后要求确认。

在不打算加密组成角色的所有文件时，ansible-vault 功能非常有用。因为指定了将要加密的文件，可以使用一个专门包含敏感数据的特殊 YAML 文件，并对其进行加密。其余角色文件仍然为普通文本，可以使用 Git 或者其他工具进行版本控制。

运行 Ansible 剧本时，在 ansible-playbook 命令中指定一个新选项（--ask-vault-pass）。例如，如果剧本文件名为 db.yml，执行的命令为 ansible-playbook db.yml --ask-vault-pass。在剧本执行之前，将要求输入 vault 密码。

11.4 角色结构优化

你可能有多个希望在所有服务器上安装的必备系统软件包，好的思路应该是创建一个基

本映像（base image）角色，部署和管理所有必备软件包（NTP、SELinux 等），而不是在每个角色中包含它们。我将创建一个名为 base 的角色，代表想在所有服务器上安装的基本映像。基本角色将包含 NTP 软件包部署指令（参见程序清单 11-8）。

程序清单 11-8　必备系统软件包所用的基本角色

```
#/etc/ansible/roles/base/tasks/main.yml
---
  - include_vars: "{{ ansible_os_family }}.yml"
  - name: Deploy NTP on Ubuntu Servers
    apt: pkg=ntp state=latest
    when: ansible_os_family == 'Debian'
  - name: Deploy NTP for Red Hat Systems
    yum: pkg=ntp state=latest
    when: ansible_os_family == 'RedHat'
  - name: NTP Service Runs
    service: name={{ ntpserver }} state=started
```

虽然 NTP 软件包名称（ntp）在 CentOS 和 Ubuntu 操作系统上相同，但服务的名称不同。所以，我们仍然需要根据操作系统家族创建不同的变量文件，如程序清单 11-9 所示。

程序清单 11-9　基本角色变量文件

```
#/etc/ansible/roles/ntp/vars/RedHat.yml
---
ntpserver: 'ntpd'

#/etc/ansible/roles/ntp/vars/Debian.yml
---
ntpserver: 'ntp'
```

现在，我们已经定义了基本角色，可以从 Web 角色中删除 NTP 部署任务，也可以创建一个新的剧本文件，部署整个 LAMP 栈（lamp.yml）。lamp.yml 文件将用于根据服务器所属组执行不同角色（参见程序清单 11-10）。

程序清单 11-10　lamp.yml

```
---
- hosts: web
  roles:
     - base
     - web
     - php
- hosts: db
  roles:
```

```
    - base
    - db
```

因为我们将所有角色放在 /etc/ansible/roles 中，lamp.yml 文件在控制器上的位置就不重要了。一旦执行该剧本，Ansible 自动知道为指定的角色检查相关的路径。

11.5 VMware 资源管理

Ansible 最近推出了 VMware vSphere 虚拟机（VM）管理支持。随着 Ansible 社区持续开发新的 VMware vSphere 管理功能，我们可能在本书的未来版本中用一整章介绍 Ansible 的 VMware 集成。现在，我们将介绍 Ansible 的 vsphere_guest 模块，在本章中讨论其功能。

vsphere_guest 模块是 Ansible 用于配给云工作负载的核心模块，支持独立 ESXi。但是，我们将把重点放在通过 VMware vCenter 管理 VMware vSphere 工作负载上。

vsphere_guest 模块依赖于开放源码项目 pysphere。所以，选择用来运行 VMware 管理任务的服务器必须通过 pip（一个 Python 软件包安装程序）手工（sudo pip install pyshphere）安装 pysphere，或者在 VMware 管理剧本中添加先决条件任务，为你安装 pysphere（参见程序清单 11-11）。

程序清单 11-11 VMware 虚拟机创建

```
- hosts: vmware
  tasks:
    - name: Deploy pip for Ubuntu Servers
      apt: pkg=python-pip state=latest
      when: ansible_os_family == 'Debian'
    - name: Deploy pip for Red Hat Systems
      yum: pkg=python-pip state=latest
      when: ansible_os_family == 'RedHat'
    - name: Deploy the pysphere Python package
      pip: name=pysphere
    - name: Deploy an Ubuntu VM on VMware vSphere
      vsphere_guest:
      vcenter_hostname: vcenter.devwidgets.io
      username: vmtrooper@devwidgets.io
      password: devopsrocks123
      guest: web server 01
      state: powered_off
      vm_extra_config:
        notes: This is a web server to run Apache in production
      vm_disk:
        disk1:
          size_gb: 10
```

```
        type: thin
        datastore: datastore1
    vm_nic:
      nic1:
        type: vmxnet3
        network: VM Network
        network_type: standard
    vm_hardware:
      memory_mb: 512
      num_cpus: 1
      osid: ubuntu64Guest
      scsi: paravirtual
    esxi:
      datacenter: Production
      hostname: vsphere01.devwidgets.io
```

在我的 Ansible 库存文件中（参见第 10 章），创建了名为 vmware 的主机组，在程序清单 11-11 中将该组指定为运行 VMware 管理任务的托管节点（-hosts: vmware）。必须限制运行 VMware 命令的主机数量，这样 vsphere_guest 才不会因为创建重复的 VM 而生成错误。可以在 vmware 组中放置单一托管节点，或者在 ansible-playbook 命令中使用 limit 选项：

```
ansible-playbook create-vm.yml --limit vmware[0]
```

vmware 组被当作一个列表，在组名之后的括号中指定列表索引。Python 的列表索引惯例在此适用。（例如，0 表示组中的第一个主机，1 表示第二个托管节点，–1 表示组中最后一个托管节点。）

程序清单 11-11 中的剧本首先满足 vsphere_guest 先决条件：验证 pip 和 pysphere 安装。正如我们用前面的 Ansible 代码所完成的，为测试环境中的不同操作系统（基于 Debian 和 Red Hat 的系统）提供任务。

最后，我们指定 vsphere_guest 指令，以便用 Ansible 创建 VM。VM 创建期间可以使用多种参数，但是我们最大限度地缩减样板代码，仅保留最有用的部分：

- vcenter_hostname：用于创建 VM 的 VMware vCenter 服务器。
- username：具有足以创建 VM 权限的用户。
- password：用户的 vCenter 密码。
- guest：创建的 VM 名称，必须是唯一的。
- state：VM 在创建之后是否应该开启电源。
- vm_extra_config：指定关于 VM 的说明，并设置 vCPU 和内存热添加等选项。
- vm_disk：VM 的存储位置。
- vm_nic：VM 连接到的 VM 网络。
- vm_hardware：指定 vCPU 数量、RAM 数量、VM 运行的 OS 等特性。

- esxi：指定创建 VM 的 VMware vSphere 数据中心和主机。

在 vm_hardware 参数中，所需操作系统的 osid 值可以从 VMware vSphere API 参考中的 VirtualMachineGuestOsIdentifier 文档获得（参见本章最后的“参考文献”）。表 11-1 展示了一些流行操作系统及对应 VirtualMachineGuestOsIdentifier 值的列表。

表 11-1 VMware vSphere 操作系统标识符

操作系统	VirtualMachineGuestOsIdentifier
Ubuntu Linux（多种版本）	ubuntu64Guest
CentOS Linux（多种版本）	centosGuest
Red Hat Linux 7	rhel7_64Guest
Windows Server 2012 R2	Windows8Server64Guest

警告 如果你用程序清单 11-1 中的代码选择另一个操作系统使用，必须验证其余 VM 参数（例如，vm_hardware）是否与之兼容。

运行程序清单 11-1 中的剧本之后，将创建 VM 并且可以在 vCenter 中使用，如图 11-1 所示。

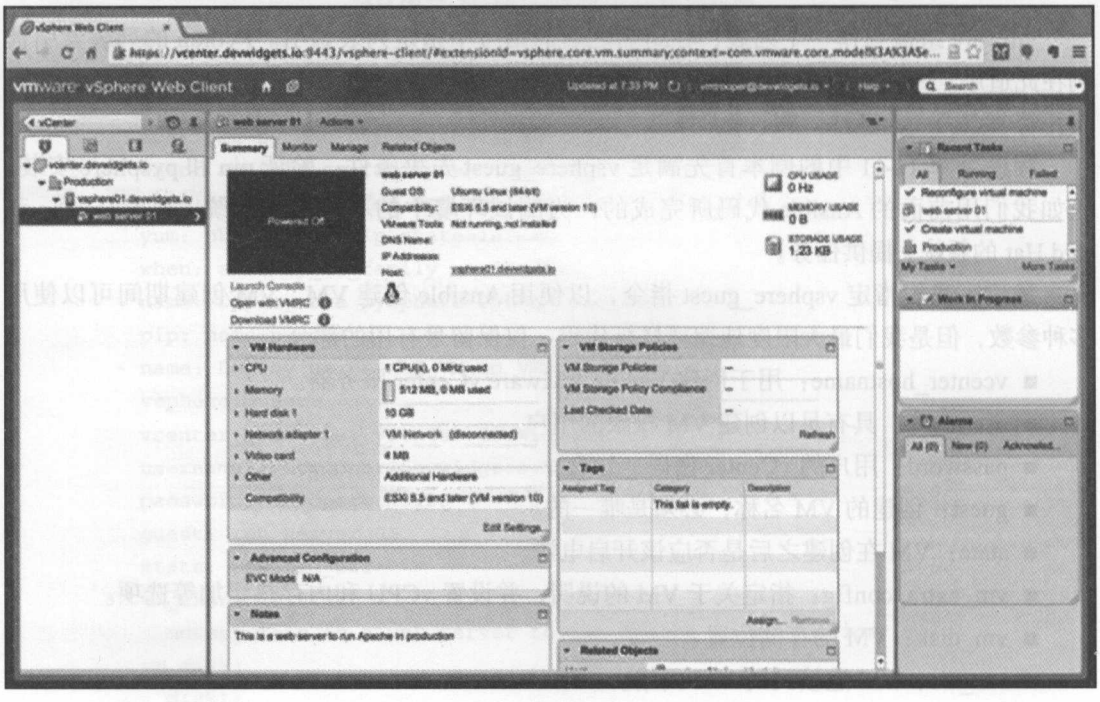


图 11-1 Ansible VMware 客户机部署结果

vsphere_guest 模块不会替你安装 VM 的操作系统。但是，你可以收集关于 VM 的事实，如 MAC 地址。可以在团队使用的任何 OS 部署工具中使用这些信息。系统管理员可以用 vmware_guest_facts 参数临时运行 vsphere_guest，获取 VM 事实：

```
ansible vmware -m vsphere_guest -a "vcenter_hostname=vcenter.devwidgets.io
username=vmtrooper@devwidgets.io password=devopsrocks123 guest='new vm001'
vmware_guest_facts=yes"
```

将 state 参数设置为 absent 可以删除 VM，将 force 参数设置为 yes 可以删除开机状态的 VM（参见程序清单 11-12）。

程序清单 11-12　VMware 虚拟机删除

```
- hosts: vmware
  tasks:
    - name: Deploy pip for Ubuntu Servers
      apt: pkg=python-pip state=latest
      when: ansible_os_family == 'Debian'
    - name: Deploy pip for Red Hat Systems
      yum: pkg=python-pip state=latest
      when: ansible_os_family == 'RedHat'
    - name: Deploy the pysphere Python package
      pip: name=pysphere
    - name: Remove a specified VM
      vsphere_guest:
        vcenter_hostname: vcenter.devwidgets.io
        username: vmtrooper@devwidgets.io
        password: devopsrocks123
        guest: web server 01
        state: absent
        force: yes
```

vsphere_guest 模块还可以修改 VM 属性，如程序清单 11-13 所示。

程序清单 11-13　VMware 虚拟机修改

```
- hosts: vmware
  tasks:
    - name: Deploy pip for Ubuntu Servers
      apt: pkg=python-pip state=latest
      when: ansible_os_family == 'Debian'
    - name: Deploy pip for Red Hat Systems
      yum: pkg=python-pip state=latest
      when: ansible_os_family == 'RedHat'
    - name: Deploy the pysphere Python package
      pip: name=pysphere
    - name: Deploy an Ubuntu VM on VMware vSphere
```

```
vsphere_guest:
  vcenter_hostname: vcenter.devwidgets.io
  username: vmtrooper@devwidgets.io
  password: devopsrocks123
  guest: web server 01
  state: reconfigured
  vm_extra_config:
    notes: This is an updated web server to run Apache in production
  vm_hardware:
    memory_mb: 1024
    num_cpus: 2
  esxi:
    datacenter: Production
    hostname: vsphere01.devwidgets.io
  force: yes
```

第一处变化是编辑 VM 状态值为 reconfigured。如果所做更改需要 VM 关机重启，可以包含 force 参数自动完成。最后，将内存（memory_mb）、vCPU 数量（num_cpus）和 VM 描述（notes）更新为所需的值。

11.6 小结

Ansible 角色可以简化应用程序部署。系统开发任务只使用必要的参数调用合适的角色。本章还讨论了如何使用 vsphere_guest 模块创建、修改、删除和获取 VMware vSphere VM 的事实。第 12 章研究 Microsoft 在 PowerShell 4.0 中为 Windows 服务器推出的预期状态（DSC）配置管理系统。

参考文献

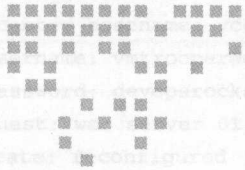
[1] Ansible documentation: http://docs.ansible.com/

[2] VMware vSphere SDK: https://www.vmware.com/support/developer/vsphere_mgmt_sdk/index.html

第五部分 Part 5

PowerShell

- 第 12 章　PowerShell 预期状态配置简介
- 第 13 章　PowerShell DSC 实施策略



Chapter 12 第 12 章

PowerShell 预期状态配置简介

PowerShell 已经快速成长为 Microsoft Windows 和 VMware 生态系统的标准自动化工具。许多 VMware 管理员利用 VMware 的 PowerShell 嵌入式管理单元 PowerCLI 执行收集环境相关信息的任务。Microsoft 专业人士已经发现，该公司投入巨资，使 PowerShell 能够管理 Microsoft Windows Server 和 Microsoft 应用程序。

到目前为止，你从本书中学到的工具是 Microsoft PowerShell 最新功能的先驱：预期状态配置（Desired State Configuration）。这一功能常常简写为 DSC，它使 Windows 管理员能够以类似于 Puppet 的功能，以原生方式管理其环境。本章介绍 DSC 的基础知识及工作原理。

本章包含如下主题：

- 什么是 PowerShell DSC
- PowerShell DSC 需求
- PowerShell DSC 组件
- PowerShell DSC 配置
- PowerShell DSC 模式
- PowerShell DSC 资源

12.1 什么是 PowerShell DSC

DSC 是 Microsoft 使 PowerShell 成为类似 Puppet 的声明性语言的第一次努力。这意味着，不再需要将完成任务所需的所有逻辑和指令编写为脚本，而是使用现有的底层框架。DSC 框架已经包含了执行请求任务的必要指令及逻辑。这并不意味着，自动化不再需要使用

熟悉的命令式编程风格，而只是另一种高可伸缩性的 Windows 环境维护方法。用户不再需要编写整个系列步骤的脚本或者程序，就能够达到预期的结果。他们只需要指示 DSC，需要对哪些系统执行哪些任务，以及相关的参数。其余事项在 DSC 框架内处理。

PowerShell 的命令式功能需求不会很快衰退。实际上，大部分管理员将继续在大部分任务（如收集信息）上以相同的方式使用 PowerShell。DSC 的采用扩展了 PowerShell 的能力，可以在显著减少所需脚本工作的情况下，应用和维护 Windows 服务器或者应用程序配置的一致性，完成这些任务的功能已经内建于框架。

DSC 可以多种方式使用，以管理 Windows 服务器环境。在分布式组织中，这些任务由操作系统（OS）团队管理，而较少由 VMware 管理员管理。当然，较小型组织可能由关键人员同时负责操作系统和虚拟环境。本书主要从 VMware 管理员的角度关注这些技术，因此，重要的是要了解开发团队可能采用的 DSC 最常见用例。

PowerShell DSC 的常见功能包含如下资源：

- 管理和监控配置状态
- 启用或者禁用 Windows 角色和功能
- 停止和启动 Windows 服务和应用程序进程
- 部署软件
- 管理
  - 环境变量
  - 文件和目录
  - 组和用户
  - 注册表设置
- 在日志中写入消息
- 运行 PowerShell 脚本（如果安装，还包括 PowerCLI）

12.2　PowerShell DSC 需求

PowerShell 4.0 默认安装于 Windows Server 2012 R2 和 Windows 8.1，也可以安装在 Windows Server 2008 R2（Service Pack 1）或者 Windows 7 Service Pack 1 上。DSC 还要求完整安装 Microsoft .NET Framework 4.5，该框架在 Windows Server 2012 R2 和 Windows 8.1 上也是默认安装的。PowerShell 4.0 向后兼容，可以运行用以前版本编写的脚本，但是 DSC 功能仅在 PowerShell 4.0 中具备。此外，PowerShell 4.0 需要 WS-Management 3.0 和 WMI 3.0，这是 Windows 管理框架（WMF）4.0 的一部分。在本书编写时，Microsoft 已经发布了 WMF 5 的预览版。虽然本书中没有研究这一版本，但是要注意，它包含了广泛的修复、优化和性能改进。

注意 在安装前一定要研究 Windows Management Framework 4.0 的附加系统需求，因为有些应用程序不兼容。如果计划使用集成脚本环境（ISE）编写 PowerShell 4.0 脚本，还要确定于 WMF 4.0 之前在 Windows Server 2008 R2（SP1）上安装 ISE。最后，在 Windows Server 2012 R2 和 Windows 8.1 上需要安装 Microsoft KB2883200。

12.3 PowerShell DSC 组件

PowerShell DSC 由多个组件构成，这些组件使其能够配置 Windows 系统。本节研究这些关键组件，帮助你认识整体的工作架构。

12.3.1 原生命令集

PowerShell DSC 首先在 Microsoft Windows PowerShell 基础上构建，因此，DSC 中内建了一组 PowerShell 命令（cmdlet）和函数，本章自始至终都将看到它们。为了访问它们，启动 PowerShell 会话，输入如下命令：

```
Get-Command -Module PSDesiredStateConfiguration
```

图 12-1 展示了 9 个选项的输出。

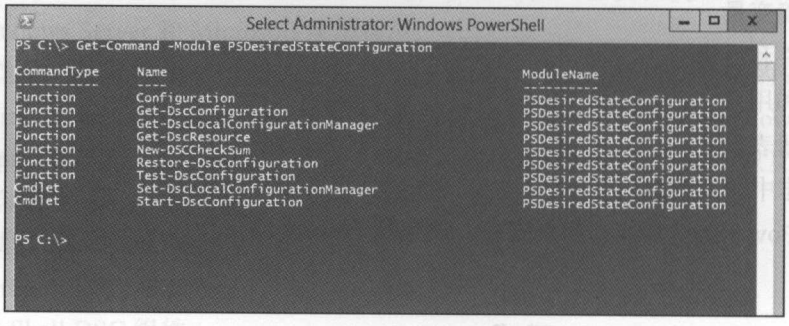


图 12-1 预期状态配置命令列表

12.3.2 托管对象格式文件

DSC 的核心是托管对象格式（MOF）文件。PowerShell 脚本通常产生这种文件，本章后面将作讨论。但是，MOF 文件可以在文本编辑器中直接编写，如果格式正确，DSC 就会接受。这对于 Microsoft 是有趣的开发方式，因为这意味着即使其他产品也有可能生成 MOF 文件，既可以用于 Windows，也可以用于 Linux 环境。实际上，在本书编写时，Microsoft 已经发布了首个用于 Linux 系统的 PowerShell DSC 的客户技术预览版本（CTP）。由于它仅是预览版本，超出了本书的范畴，但是在考虑 DevOps 组织如何利用多种工具交付资源时，这是需要

注意的有趣发展。Microsoft 明显致力于使 PowerShell DSC 适用于广泛的环境和用例。

> **注意** DSC 使用的 MOF 文件路径可以是本地路径，也可以是 UNC 路径，这对于想要集中管理 MOF 文件的团队极有价值，尤其是使用“拉”模式的团队。如果你打算使用 UNC 路径，节点必须能够访问该路径。

12.3.3　本地配置管理器

本地配置管理器（LCM）是每个启用 DSC 的系统上的本地引擎，用于调用配置脚本中的资源。LCM 不仅是 DSC 架构中的关键组件，而且可以为每个节点的独特需求定制。这些属性可以描述 LCM 的行为，如检查更新的频率、找到新配置时的行为、节点需要运作于“推”或者“拉”模式等。Set-DscLocalConfigurationManager 命令用于应用自定义的设置。

因为 DSC 提供了设置 LCM 的选项，了解 Get-LocalConfigurationManager 命令能够为你提供适用于 LCM 的系统当前设置就很重要了，如图 12-2 所示。图中展示了 Microsoft Windows 2012 系统上的默认设置。

```
PS C:\Users\Administrator> Get-DscLocalConfigurationManager


AllowModuleOverwrite            : False
CertificateID                   :
ConfigurationID                 :
ConfigurationMode               : ApplyAndMonitor
ConfigurationModeFrequencyMins  : 30
Credential                      :
DownloadManagerCustomData       :
DownloadManagerName             :
RebootNodeIfNeeded              : False
RefreshFrequencyMins            : 15
RefreshMode                     : PUSH
PSComputerName                  :
```

图 12-2　DSC 本地配置管理器默认设置

我们可以用程序清单 12-1 中的简短例子对 LCM 配置进行初步更改。在此，我们简单地更改 LCM 的几个选项，并在图 12-3 中展示系统的变化。我们提供了一个注释，说明脚本中应用配置信息的位置，这使开发团队或者系统管理员可以同时应用 LCM 配置和必要的节点设置。

程序清单 12-1　对 LCM 配置进行初步修改

```
Configuration LCMExample
{
    Node "localhost"
    {
     LocalConfigurationManager
      {
       ConfigurationModeFrequencyMins = 45
       ConfigurationMode = "ApplyAndAutocorrect"
       RebootNodeIfNeeded = $true
       AllowModuleOverwrite = $false
      }
     # Resource blocks added below when needed
    }
}
```

```
# Apply the configuration by entering the following in PowerShell
$path = "C:\Users\Administrator\Documents\DevOps"
LCMExample -OutputPath $path
Set-DscLocalConfigurationManager -Path $path
```

```
PS C:\Users\Administrator> Set-DscLocalConfigurationManager -Path "C:\Users\Administrator\Documents\DevOps"
PS C:\Users\Administrator> Get-DscLocalConfigurationManager

AllowModuleOverwrite            : False
CertificateID                   :
ConfigurationID                 :
ConfigurationMode               : ApplyAndAutoCorrect
ConfigurationModeFrequencyMins  : 45
Credential                      :
DownloadManagerCustomData       :
DownloadManagerName             :
RebootNodeIfNeeded              : True
RefreshFrequencyMins            : 15
RefreshMode                     : PUSH
PSComputerName                  :


PS C:\Users\Administrator>
```

图 12-3 设置 DSC 本地配置管理器配置

注意 Microsoft Windows PowerShell 集成脚本环境（ISE）是构建 DSC 配置的出色工具，它包含在 WMF4.0 安装中，可以帮助你构建自己的配置。你可以在必要时编写、运行和保存配置，记住，ISE 的 32 位版本（x86）不支持 PowerShell DSC 配置。

应用前面概述的 DSC LCM 配置的结果是指定路径下名为 < 主机名 >.mof 的 MOF 文件。该文件的文件名可能是 localhost.mof，这取决于 Node 部分开始处的定义。在我们的例子中，输出文件将为 localhost.mof，因为我们将 Node（节点）定义为 localhost。

12.4 PowerShell DSC 配置

PowerShell DSC 配置就是脚本，格式类似于函数，用于在 PowerShell DSC 中运行，创建系统 / 节点使用的 MOF 文件。

这些配置可能非常复杂，但是可以通过 3 个主要部分理解它们：

- 容器
- 节点声明
- 配置

程序清单 12-2 中的简单示例说明了必要的基本组件。

程序清单 12-2 必要的基本组件

```
Configuration Test01
{
    param ($NODENAME="localhost")
    Node $NODENAME
```

```
    {
     WindowsFeature IIS
     {
      Ensure = "Present"
      Name = "Web-Server"
     }
    }
}
```

在这个例子中，我们建立了一个基本配置，用于在 Windows Server 2012 中安装 IIS。我们将配置命名为 Test01，默认将节点设置为本地主机。因为我们为 $nodename 声明了一个参数，也可以传递系统名，用这个配置为任何系统创建 MOF 文件：

```
Test01 YourNode
```

上述命令为提供的节点名称创建 MOF 文件。PowerShell 也可以使用多个参数，多参数的用例包括复制文件和选择目录及注册表位置。

如果我们将 LCM 配置与附加的资源块组合起来，就可以开始形成更完整的配置脚本，如程序清单 12-3 所示。在这个配置中，我们将节点名称作为参数，用特定的配置需求配置 LCM 并提供将节点带入最终的预期状态所需的资源，保持重用的机会。

程序清单 12-3　LCM 配置示例

```
Configuration LCMExample
{
    param ($NODENAME="localhost")
    Node $NODENAME
    {
     LocalConfigurationManager {
        ConfigurationModeFrequencyMins = 45
        ConfigurationMode = "ApplyAndAutocorrect"
        RebootNodeIfNeeded = $true
        AllowModuleOverwrite = $false
        }
     WindowsFeature IIS {
        Ensure = "Present"
        Name = "Web-Server"
        }
     File Scripts {
        Ensure = "Present"
        Type = "Directory"
        SourcePath = "\\scriptserver\Scripts\"
        DestinationPath = "C:\Users\Administrator\Documents\Scripts"
        Recurse = "True"
        }
```

```
    }
}
```

如果你已经阅读了前面几章，就可能已经注意到 DSC 配置文件的格式类似于 Puppet，每个配置资源单独添加。和 Puppet 一样，可以多次利用每个资源。

12.5 PowerShell DSC 模式

DSC 设计为以两种不同模式运作："推"和"拉"。但是，将 DSC 看作 3 种模式可能更容易：本地推送、远程推送和拉取。每个选项都有特定的优势和不足，但是它们都是 DSC 完整能力的一部分。

12.5.1 本地推送模式

本地推送模式是使用 PowerShell DSC 的最简单方法。在这种方法中，管理员只是登录系统，本地运行配置脚本，本地初始化配置。在 12.4 节中的例子很容易在节点本地实现。注意，在图 12-4 中 DSC 使用的所有组件都在节点上维护。

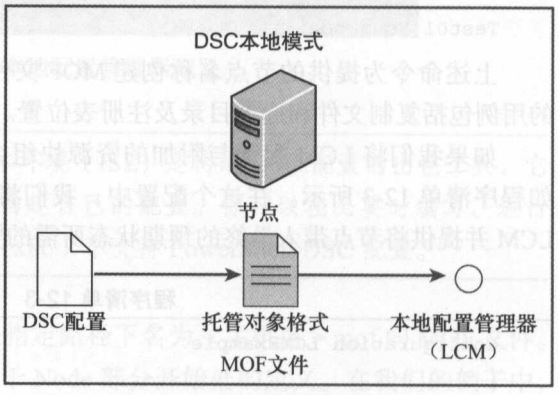


图 12-4 DSC 本地推送模式

12.5.2 远程推送模式

第二种模式——推送模式也很容易应用，但是有额外的要求和注意事项。在这种模式下，管理员将运行配置脚本，对远程节点或者图 12-5 中概述的节点执行配置。因为 PowerShell 在用，可以对多个节点应用相同的配置调用，生成它们各自的 MOF 文件并投入使用。这提供了更好的伸缩性，特别是在经常部署多层应用程序的敏捷环境中。

推送模式的缺点之一是必须在每个目标节点上启用 PowerShell 远程处理。对推送模式感兴趣的人们必须对每个系统运行 Enable-PSRemoting 命令，或者将其作为虚拟机（VM）模板的一部分。为了使推送模式有效，DNS 应该是当前的。建议在所有节点处于同一个域的环境中使用推送模式，因为 PowerShell 远程处理更容易配置，一旦节点处于该域，DNS 应该保持最新。

远程执行 DSC 配置还要求管理员的 PowerShell 会话以管理员模式运行，可以通过观察 PowerShell 会话标题栏识别是否处于管理员模式。如果目前没有处于这种提升模式，右键单击 PowerShell 快捷方式，选择"以管理员身份运行"，可以启动新的管理员模式会话。

还要注意，使用推送模式时，目标节点必须本地调用 DSC 资源。如果目标节点没有必要的资源，配置无法应用，你可能看不到任何辨别失败原因的错误。正如 12.6 节中所描述

的，可以用 Get-DscResource 命令检查这些资源的可用性。

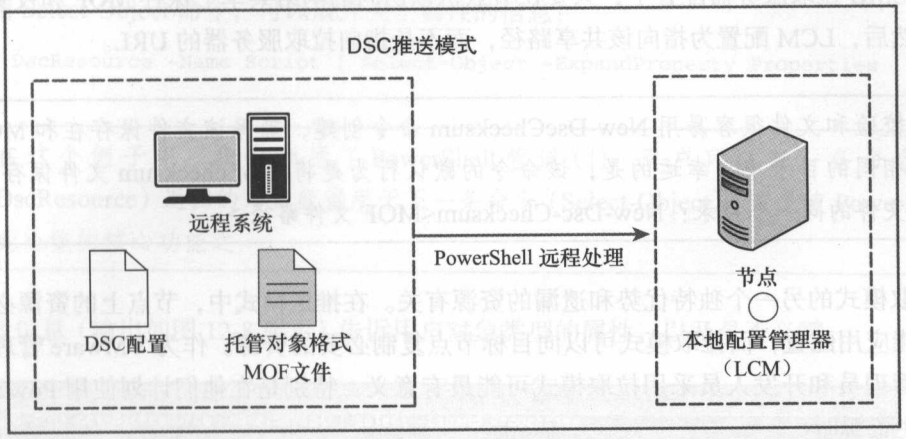


图 12-5　DSC 远程推送模式

12.5.3　拉取模式

拉取模式是最为复杂的模式，但是也最为灵活。拉取模式利用运行 DSC 服务的中心服务器，节点的本地配置管理器（LCM）从中寻找其 MOF 文件，如图 12-6 中的框图所示。拉取模式可以使用两种交付机制：HTTP/HTTPS 或者 SMB。遗憾的是，拉取服务器的配置相当麻烦。Microsoft 和多位社区贡献者已经提供了资源，可以用 DSC 自身配置 HTTP 拉取服务器！这些资源安装 IIS 和其他相关服务，创建必需的目录，并提供必需的配置文件。

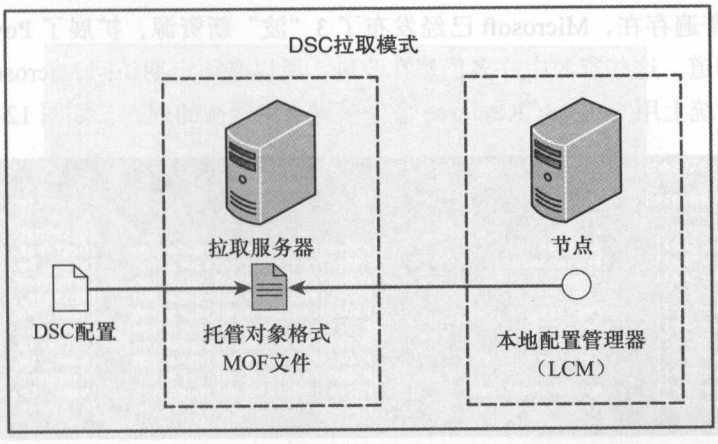


图 12-6　DSC 拉取模式

如果组织选择，可以用 HTTPS 设置拉取服务器。常见用例之一是你希望用 DSC 维护配置的系统处于非军事区（DMZ）时。为拉取服务器使用 HTTPS 需要在节点上安装受信任的

证书，才能够请求配置信息。

在 SMB 拉取服务器配置下，只要在节点可以访问的网络共享上保存 MOF 和校验和文件即可。然后，LCM 配置为指向该共享路径，而不是指向拉取服务器的 URL。

校验和文件很容易用 New-DscChecksum 命令创建，确保该文件保存在和 MOF 文件相同的目录中。幸运的是，该命令的默认行为是将 .mof.checksum 文件保存在 MOF 文件的同一个目录：New-Dsc-Checksum<MOF 文件路径 >。

拉取模式的另一个独特优势和遗漏的资源有关。在推送模式中，节点上的资源必须预先存在才能应用配置，而拉取模式可以向目标节点复制必要的资源。作为 VMware 管理员，鼓励系统管理员和开发人员采用拉取模式可能最有意义，特别是在他们计划使用 PowerShell 4 的非原生资源时。

最后，拉取模式在组织对监控和维护依从性感兴趣时很理想。你可以配置节点的行为，检查节点的 LCM，在本章后面将作讨论。这些设置包括扫描更改的频率，以及发现更改时的行为。

12.6 PowerShell DSC 资源

寻求完全利用 DSC 功能的管理员和开发人员将会调查可用的配置资源。这些资源提供了 DSC 中固有的框架。这些提供者的强大之处在于将多个资源组合为节点的配置。

本书前面已经提到过，PowerShell DSC 在安装 WMF 4.0 时启用，具备通用配置资源。由于 WMF 4.0 普遍存在，Microsoft 已经发布了 3 “波” 新资源，扩展了 PowerShell DSC 的原生能力。要知道，这些资源中许多仍然在改进，所以必须定期访问 Microsoft TechNet 获取更新。可以在系统上用 Get-DSCResource 命令获得可用资源的列表，如图 12-7 所示。

```
PS C:\Users\jatwell> Get-DscResource

ImplementedAs  Name            Module                          Properties
-------------  ----            ------                          ----------
Binary         File                                            {Destinat...
PowerShell     Archive         PSDesiredStateConfiguration     {Destinat...
PowerShell     Environment     PSDesiredStateConfiguration     {Name, De...
PowerShell     Group           PSDesiredStateConfiguration     {GroupNam...
Binary         Log             PSDesiredStateConfiguration     {Message,...
PowerShell     Package         PSDesiredStateConfiguration     {Name, Pa...
PowerShell     Registry        PSDesiredStateConfiguration     {Key, Val...
PowerShell     Script          PSDesiredStateConfiguration     {GetScrip...
PowerShell     Service         PSDesiredStateConfiguration     {Name, Bu...
PowerShell     User            PSDesiredStateConfiguration     {UserName...
PowerShell     WindowsFeature  PSDesiredStateConfiguration     {Name, Cr...
PowerShell     WindowsProcess  PSDesiredStateConfiguration     {Argument...
```

图 12-7 Get-DSCResource 命令

PowerShell DSC 新用户所面对的挑战之一是理解与每个 DSC 资源相关的属性。Microsoft 保持着在 PowerShell 会话中可获得宝贵信息的传统。有两种途径可以获得与资源属性有关的

附加信息。下面的例子回顾了 Script 资源的属性和语法，在前面的图 12-7 中已经列出。

使用 Select-Object 命令，可以展开关于属性的信息：

```
Get-DscResource -Name Script | Select-Object -ExpandProperty Properties
```


注意　在这个例子中，我们利用了 PowerShell 管道（|），用户可以将一条命令（Get-DscResource）返回的对象数据用于下一条命令（Select-Object），这是使 PowerShell 如此全能的核心功能之一。

这些信息（输出如图 12-8 所示）告诉用户对象类型的属性，以及是否必需。

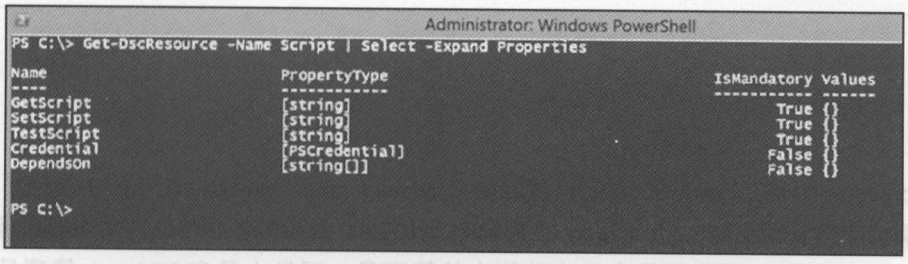


图 12-8　Select-Object 命令

在 Get-DscResource 命令中使用 -Syntax 参数，可以显示在 PowerShell 配置脚本中必须使用的格式，如图 12-9 所示：

```
Get-DscResource -Name Script -Syntax
```

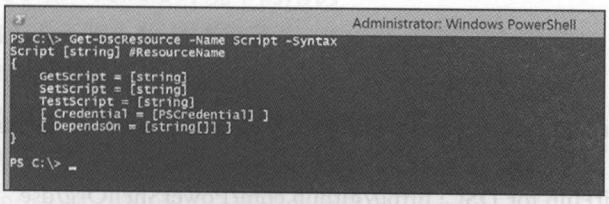


图 12-9　Get-DscResource 命令的 –Syntax 参数

注意，在配置脚本中添加资源时，必须用指定的资源名称代替 #ResourceName。

许多新提供者重点关注 Microsoft Hyper-V、SQL 和 Active Directory。虽然第一项对普通 VMware 管理员来说没有太大用处，但是依赖于 Windows 平台的 DevOps 团队将会尽快实施后面两项。

PowerShell 社区已经致力于制作可以加入配置的其他资源。最引人注目的是，应该访问 PowerShell.org GitHub，在那里可以找到 DSC 的深入研究和社区生成的资源。因为这些资源是社区构建的，它们没有得到官方支持，在使用它们（尤其是生产环境中）之前应该进行评估。

可以确信的是，Microsoft 正在迅速行动，通过资源工具包发布新的 DSC 提供者。在一年内已经发布了 9 波资源工具包，预计 Microsoft 和其他供应商还将发布更多的资源，可以肯定地说，开发团队将随着 PowerShell DSC 成熟更好地利用它。唯一的难题是，当前发布的资源不总是能得到保证，应该根据现状处理。当前未受支持的资源很容易识别，因为它们的名称将带有 x 前缀。

注意 Microsoft 说明，资源工具包应该在 Windows 8.1 和 Windows Server 2012 R2 上安装。在运行旧操作系统的系统上添加资源工具包和社区提供者时可能有额外的要求。我们已经在本章中尝试提供这些先决条件，但是建议在实施和使用这些工具包之前阅读所提供的发行说明。

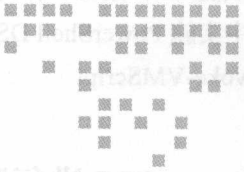
12.7 小结

Microsoft 已经在 PowerShell 4 和 Windows Management Framework 4.0 中构建了非常灵活的预期状态平台。Microsoft Windows PowerShell DSC 提供根据以 MOF 文件形式部署的配置动态配置 Windows 系统的能力。这种能力使管理员、开发人员和 VMware 管理员可以预先定义 Microsoft Windows 系统的特性，随时维护其配置。这在 Linux 环境中不是独特的功能（可以使用前几章描述的 Puppet、Chef 和 Ansible），但是对于 Windows 环境是巨大的进步。DSC 的开发仍处于早期阶段，管理员可以预期在几个月内从 Microsoft 得到更强大的功能，远远超过本书中概述的能力。

参考文献

[1] Microsoft TechNet DSC resources: http://technet.microsoft.com/en-us/library/dn282125.aspx

[2] PowerShell.Org GitHub for DSC: http://github.com/PowerShellOrg/dsc



第 13 章 Chapter 13

PowerShell DSC 实施策略

VMware 管理员在 DevOps 环境中处于独特的地位，因为他们在运营和开发方面的参与度很高。这种地位要求他们在向开发团队交付虚拟机（VM）和虚拟基础设施时必须敏捷。许多 VMware 管理员使用 VMware 的 PowerShell 内嵌管理单元 PowerCLI 自动化 VMware vSphere 环境任务和报告。本章讨论 VMware 管理员使用 PowerCLI、PowerShell 4.0 和 PowerShell 预期状态配置（DSC）服务整个 DevOps 团队需求的方法。

本章包含如下主题：

- PowerShell DSC 在 VMware 环境中的用例
- 使用 PowerCLI 进行脚本化 VM 部署
- 在 VM 模块中加入 PowerShell DSC
- 对新 VM 实施 PowerShell DSC 配置所面临的挑战
- 经验教训总结
- PowerShell DSC 未来在 VMware 环境中的用例

13.1 PowerShell DSC 在 VMware 环境中的用例

在本书编著时，PowerShell DSC 没有发布任何用于管理 VMware 基础设施的资源。Microsoft 已经为 Hyper-V 的管理发布了几个初步资源，将在本章后面进一步讨论。今天的 VMware 管理员可以使用多种方法对所管理环境中的 VM 操作系统应用 DSC 配置。本节概述这些方法的不同策略和用例。后面的小节介绍实施这些方法所面临的一些挑战，以便帮助你在自己的环境中实施之前理解这些挑战：

- 对 VM 实施 DSC 配置，作为 PowerCLI 部署脚本的一部分
- 在模板中实施 PowerShell DSC
- 使用 Invoke-VMScript

13.2 用 PowerCLI 进行脚本化 VM 部署

PowerCLI 是 VMware 的 PowerShell 嵌入式管理单元，VMware 管理员可以用它的 New-VM 命令部署新 VM。这条强大的命令可以从各种出发点构建 VM。最常见的是，在成熟的 VMware 环境中利用模板和操作系统自定义规格构建 VM，如程序清单 13-1 所示。

注意 VMware OS 在自定义 New-VM 的部署中不是必要的，但是提供了部署 VM 的系统化方法。操作系统许可证、域和网络配置等信息可以通过使用 OS 自定义规格实施。

程序清单 13-1 创建 VM

```
###### Creating the VM #####

# The following information is required for connecting to the vCenter
# Server
$vcenter = "vTesseract-vc51"

# $vcenter = Read-Host 'What is your vCenter Server Name?'

# $cred = Get-Credential
<# We recommend you never put credentials in plain text.
Use the above $cred call when running the script interactively.
Specifying credentials is NOT required when the script is being run by a
user whose logged in credentials match credentials with appropriate vCenter
permissions assigned.
#>


<# Collect or Define VM Info.
 In this section I have defined values.
 You can opt to collect info yourself from environment
 or user VM Info
#>

$VMName = 'devops-02'
# Read-Host 'VM Name?'

# VM Deployment Requirements
```

```
$TemplateName = "w2k12r2-01"
# Read-Host 'Template Name?'

$OSSpecName = "W2K12R2"
# Read-Host 'OS Customization Spec Name?'

$DatastoreCluster = "vTesseract-Nas01"
# Read-Host 'Datastore Cluster Name?'

$VMhostCluster = "MGMT"
# Read-Host 'Cluster Name?'

$FolderName = "Devops"
# Read-Host 'Folder Name?'

###### Script Execution Section ######

# Connect to vCenter Server
if($cred -eq $null){
  Connect-VIserver $vcenter
}else{
  Connect-VIserver $vcenter -Credential $cred
}

# Compile New VM Info
$Template = Get-Template -Name $TemplateName
$OSSpec = Get-OSCustomizationSpec $OSSpecName
$Datastore = Get-DatastoreCluster -Name $DatastoreCluster
# Datastore Cluster assumes Storage DRS is turned on for the Cluster
$Cluster = Get-Cluster -Name $VMhostCluster
$Folder = Get-Folder $FolderName

##### Creates the new VM based on collected info. #####

New-VM -Name $VMName -Template $Template -OSCustomizationSpec $OSSpec '
 -Datastore $Datastore -ResourcePool $Cluster -Location $Folder
```

在程序清单 13-1 中，我已经在可以使用脚本提示用户输入、以实现更好交互性的地方做了注释。要使用它们，只需要如程序清单 13-2 中所示的例子，将该变量的值设置为等于 Read-Host 语句。

程序清单 13-2　用户提示示例

```
###### Creating the VM #####
```

```
# The following information is required for connecting to the vCenter
# Server
$vcenter = Read-Host 'What is your vCenter Server Name?'
```

你可能注意到，程序清单 13-1 中脚本的大部分涉及调用 New-VM 的最后一行必需信息的收集。应该使用 Get-Help New-VM -Full 获得 New-VM 命令部署 VM 的更完整介绍。

13.3 在 VM 模板中加入 PowerShell DSC

在第 12 章中学习了配置本地配置管理器（LCM）指向中央拉取服务器的方法。VMware 管理员可以和操作系统管理员或者开发团队合作，确保 VM 模板预先配置 LCM，指向特定的拉取服务器。这为 DSC 拉取服务器管理员在描述所部署 VM 特性时提供了灵活性。

此外，这种做法还使 VMware 管理员更容易保持 VM 模板最新。正确实施 Windows 服务器更新服务（WSUS）并将 PowerShell DSC 指向中央拉取服务器，意味着保持模板最新更简单、更一致。维护正确补丁和更新的模板，确保部署之后的配置和所需补丁最少。减少新 VM 可用所需的时间是高效 DevOps 活动的基础。

使用具备拉取服务器的 PowerShell DSC 的局限性之一是要求新 VM 可以通过网络与服务器通信。采用隔离开发环境或者在 DMZ 中部署的组织可能觉得成功使用 DSC 管理这些 VM 很困难。支持这些环境的最简单机制是在同一网络上提供拉取服务器，或者通过防火墙设置一个开放路径。因为这会带来潜在的风险，我们已经努力寻找在不需要与隔离区之外通信的情况下加入 DSC 的方法，这将在后面讨论。

13.4 对新 VM 实施 PowerShell DSC 配置所面临的挑战

在前面，我们讨论了在 VMware 虚拟环境中实施 DSC 的各种不同方法。这些不同机制的主要前提是专注于最小化部署时间，并在网络分段使集中 DSC 管理难以进行的环境下提供最大的灵活性。

虚拟网络的改进使网络隔离和管理更加灵活；但是，探索传统拉取服务器架构的替代方案，以了解用其他 VMware 独特机制能否有效地实施 PowerShell DSC 是值得的。注意，我们指出的所有难题都不会打断实施，只是需要预先计划，根据环境限制做出妥协。用 PowerShell DSC 维护 VM 的最有效方法仍然是使用拉取服务器。

注意 前面提示的方法提供了不同的结果一致性水平，最终证明在大部分情况下没有太多益处。在决定采用这些方法之前考虑你的用例，因为列出的局限性可能使这种方法不适合于你的需求。

13.4.1　PowerCLI Invoke-VMscript

VMware 可以利用 VMware 工具在 PowerCLI 中启用 Invoke-VMscript 命令。这条命令使管理员可以通过 ESXi 主机与 VM 上安装的 VMware Tools 通信，在 VM 上运行一个脚本。当然，如果将 Invoke-VMscript 和 DSC 组合使用，有一些要求。

1. VM 必须开启电源。
2. VMware Tools 必须安装并运行。
3. 系统上必须安装 PowerShell。
4. 运行脚本的用户必须有 VM 所在文件夹的访问权限。
5. 运行脚本的用户必须有 Virtual Machine.Interaction.Console Interaction 权限。
6. ESXi 主机端口 902 必须打开。
7. 用户必须能够通过 ESXi 主机和 VM 客户操作系统的身份验证。
8. 客户操作系统也必须安装 WMF 4.0 和 PowerShell 4.0。

上述需求相对简单，可以包含在一个脚本中。不建议在脚本中包含以普通文本显示的凭据，这可以在脚本中用程序清单 13-1 中使用过的 Get-Credential 命令避免。这条命令将提示执行脚本的人输入凭据，并以加密格式保存应答。

遗憾的是，Invoke-VMscript 有一些局限，大大地降低了用于网络隔离 VM 的效能。主要的局限是该命令的脚本参数似乎只接受单行脚本。这对于许多脚本活动很合适，但是不足以正确实施 PowerShell DSC。如果 VM 可以从网络上访问拉取服务器或者网络共享上的配置脚本，对 VMware 管理员来说，独特优势就很有限。

这并不意味着必须放弃 Invoke-VMscript。可以提供 PowerShell 脚本文件的完整路径，使用同时安装了 PowerCLI 和 PowerShell 的系统，就可以生成一个 MOF 文件，将其保存在可访问的位置，并使用 Invoke-VMscript 初始化配置，如程序清单 13-3 所示。这段代码应该在添加 DSC 配置脚本 webserver 并指定生成 MOF 文件的网络位置之后加入。

程序清单 13-3　创建 MOF 文件

```
# Creating MOF file
$mofpath = "\\<server>\moffiles\$vmname\"
WebServer $VMName -OutputPath $mofpath

$ScriptText = "Set-DscLocalConfigurationManager -Path $mofpath
-ComputerName $VMName"

# Get the VM object from vCenter
$vm = Get-VM $VMName

# Prompt user for VM guest OS credentials
$guestcred = Get-Credential

# Invoke the script on the Guest OS
```

```
Invoke-VMscript -ScriptText $ScriptText -VM $vm -GuestCredential
$guestcred -ScriptType PowerShell -Server $vcenter
```

诚然，上述方法和使用远程推送模式很类似。但是，因为我们通过 Invoke-VMscript 初始化 DSC 配置脚本，不需要在目标 VM 上启用 PowerShell 远程处理。使用这种技术的测试结果各不相同，偶尔会有远程系统拒绝 MOF 文件的情况，且该问题的根源尚不确定。

13.4.2 PowerCLI Copy-VMGuestFile

如果网络隔离是绝对的要求，仍然有可能利用 Invoke-VMscript 首先将 MOF 文件或者脚本复制到 VM。这可以使用 PowerCLI 中的 Copy-VMGuestFile 命令完成。这条命令可以通过 VM 安装的 VMware Tools 从执行系统中向 VM 文件系统复制一个文件。根据所运行的 VMware vSphere 版本，这条命令有不同的要求：

1. VM 必须开启电源。
2. VMware Tools 必须安装和运行。
3. 系统上必须安装 PowerShell。
4. 运行脚本的用户必须有 VM 所在文件夹的访问权限。
5. 运行脚本的用户必须有 VirtualMachine.GuestOperations.Modify 权限。
6. ESXi 主机端口 902 必须打开。
7. 用户必须能够通过 ESXi 主机和 VM 客户操作系统的身份验证。
8. 该命令必须能够支持客户操作系统。

要实施 DSC，必须首先将配置脚本或者 MOF 文件复制到 VM，然后按照前面的描述使用 Invoke-VMscript cmdlet，程序清单 13-4 是一个简化的示例。这个脚本在前面的例子基础上构建，假定你已经创建了一个脚本文件，包含配置、构建 MOF 文件和执行配置。

程序清单 13-4 通过 Invoke-VMscript 实施 DSC

```
$script = "E:\Scripts\DevOps\WebServer.psd1"
$mofpath = "C:\Users\Administrator\Documents"
$moffile = "C:\Users\Administrator\Documents\WebServer.psd1"
# The following line invokes the configuration and creates a file called
Server001.meta.mof at the specified path
$guestcred = Get-Credential
$BuildMof = WebServer -OutputPath $mofpath

# Get the virtual machine object from vCenter server
$vm = Get-VM devops-02
# Copy the script to the designated directory on the target VM
```

```
Copy-VMGuestFile -Source $script -Destination $mofpath -LocalToGuest -VM
$vm -HostCredential $cred -Server $vcenter -GuestCredential $guestcred
# Invoke the copied script on the target VM
Invoke-VMScript -ScriptText $moffile -VM $vm -HostCredential $cred -Server
$global:DefaultVIServer -GuestCredential $guestcred
```

这种方法使 VMware 管理员可以在与网络隔离，或者因为安全考虑无法支持 PowerShell Remoting 的 VM 上执行 DSC 配置。

注意　已经证明，上述方法在我们设置的测试条件下是最一致、最高效的。因为要求和简单使用 Invoke-VMscript 几乎完全相同，我们建议在需要保持 VM 隔离或者避免启用 PowerShell 远程功能时使用 Copy-VMGuestFile。

13.5　经验教训总结

我们承认，在上述用例中，有些本身不是理想的方案。但是，重要的是理解 VMware 管理员可以自由支配的各种工具。Invoke-VMscript 和 Copy-VMGuestFile 都有局限性，可能会阻碍你对它们的利用。正确地评估和证明你的需求，以便选择最有效的方法，是很重要的。

在测试中，我们发现没有一种方法能够达到 100% 的一致。这通常是由于人为错误，因为我们所要做的是以前没有尝试过的事情。尽管如此，本章介绍的各种方法在合适的条件下都能取得成功。我们的意图是，在你也感觉需要采用这些方法在无法使用原生方法的系统上利用 PowerShell DSC 时，这些努力能提供某种指导。

13.6　未来 PowerShell DSC 在 VMware 环境中的用例

管理员将继续使用 PowerShell 执行广泛的配置、管理和报告任务。DSC 可以扩展，远远超出 Windows 系统和 Microsoft 应用程序的配置。如前所述，Microsoft 已经展示了使用 DSC 管理 Linux 系统的能力。可以肯定，这些功能将扩展到 VMware 生态系统中。

定制 PowerShell DSC 资源当然是一种可能性。以特定格式提供模式，可使资源能够创建正确格式化的 MOF 文件。和所有 PowerShell DSC 资源一样，这些 MOF 文件将指示资源达到正确的最终状态。资源将包含 PowerShell 模块和实施更改的必要代码。这种格式化需求超出了本书的范围，但是我们可以参考 Microsoft 的 TechNet 网站。

PowerShell DSC 的全部可扩展性尚未得到彻底的研究，因为它仍然是 PowerShell 和 Microsoft Windows 管理中相对新颖的功能。可以肯定地说，随着这一功能的成熟，我们将看到它的功能也得到很大的扩展。如果在 PowerShell DSC 中看到复制当前 VMware 特性（如

主机配置文件和配置管理器）的功能，应该不会令人吃惊。此外，VMware 管理员可以关注，在不远的将来也许能看到用 PowerShell DSC 维护 VM 硬件配置文件、文件夹结构或者网络配置的功能。

最后，围绕 VMware 数据中心的基础设施组件生态系统继续发展，并产生更为灵活的应用程序编程接口（API）和集成点。我们可以想象这样一种情景：在不远的将来，PowerShell DSC 及其功能将为下一代软件定义数据中心的效率带来巨大的贡献。存储、网络和计算机组件的可扩展性将会得到很多机会，改善整个基础设施栈配置标准的维护方式。

注意 Microsoft 定期发行新的 PowerShell DSC 资源。在环境中实施它们之前应该注意，这些资源往往只有有限的支持，应该据此使用。

13.7 小结

现在，你应该更好地理解在 DevOps 环境中协助实施 PowerShell DSC 的途径，尽管它有着潜在的局限性挑战。大部分 VMware 管理员对于本章中概述的方法需求可能相当有限。

作为 PowerShell 4 和即将面世的 PowerShell 5 中的一项功能，随着资源库以很快的速度制作，DSC 将继续成长。Microsoft 努力地推动 Hyper-V 资源的加入，可以合理地预期，利用 PowerShell 的可扩展性，VMware 和其他公司最终将随之成为合适的用例。不管时间表如何，PowerShell DSC 很快会成为以 Microsoft Windows 为中心的 DevOps 环境中的主流。作为这种环境下的 VMware 管理员，重要的是深入了解 DSC，既能够很好地阐述它，又能为成功实施做出贡献。

参考文献

[1] VMware developer portal for PowerCLI cmdlet `Copy-VMGuestFile`: https://www.vmware.com/support/developer/PowerCLI/PowerCLI55/html/Copy-VMGuestFile.html

[2] Microsoft TechNet DSC custom resources: http://technet.microsoft.com/en-us/library/dn249927.aspx

第六部分 *Part 6*

利用容器进行应用程序部署

- 第 14 章　Docker 应用容器简介
- 第 15 章　大规模运行 Docker 容器

Chapter 14 第 14 章

Docker 应用容器简介

本章介绍一种越来越流行的应用程序部署和管理方法——使用 Docker 驱动的 Linux 容器。首先，我们介绍应用程序的定义以及它们难以管理的原因。接下来，我们了解 Linux 容器的实际工作原理，然后研究 Docker 及其产生可移植、自包含的应用容器，以快速而可重复的方式部署的能力。

本章包含如下主题：

- 什么是应用程序？
- Linux 容器
- Docker 的使用

14.1 什么是应用程序

应用程序是设计用于提供一组特性或者服务的软件逻辑单元。大部分软件帮助人们执行特定任务（如获得前往附近餐厅的路线，或者使用 140 个字符以内发表热门话题的意见）。

表面上，用户将应用程序视为一个简单的接口，但是作为系统管理员，我们知道应用程序或者其执行环境内部隐藏的复杂性。

14.1.1 隐藏的复杂性

所有的应用程序都有依赖性。作为系统管理员，我们通常与外部依赖性（如第三方软件包、语言运行时库、系统库和数据库）交互。

每个依赖关系通常都带来各种配置选项，如数据库用户名和密码定义。前几章讨论的许

多工具集中于应用程序、应用程序相互依赖性和相关配置的自动化，但是应用程序管理真正复杂的地方在于配置和依赖性的相互冲突。这些冲突就是我们无法在同一个服务器上运行某个应用程序多种版本的原因。

14.1.2　依赖性和配置冲突

冲突发生在两个应用程序需要不同的系统库或者相互冲突的语言运行时库的情况下。

在系统库冲突的情况下，我们通常会发现将要安装的一个应用程序版本需要特定版本的库。这发生在应用程序根据库的特定版本编写时，操作系统供应商为了实现长期稳定性，往往鼓励这种做法。但是，这样做是有代价的，当你想要开发需要同一个库新版本的新应用程序时怎么办?

权宜之计可能非常复杂，以至于完全不可能实现。例如，以不同路径和名称同时安装库的新版本和旧版本是一种变通的办法：

- /usr/lib/openssl.so
- usr/lib/openssl-1.0.1.so

这种变通方法可能获得成功，但是我们必须配置应用程序，使其知道这一差异。

依赖性管理问题的真正解决方案就是隔离。

14.2　Linux 容器

Linux 容器提供应用程序之间的必要隔离，以消除运行时依赖性和配置之间的冲突。隔离还提供了应用程序多个版本同时运行的途径。这一概念对于零停机时间部署和回滚到前一应用程序版本来说是强有力的。

使用容器，可以将应用及其依赖性组合为一个包，成为带有版本的工件。需要理解的是，容器并不提供完整的虚拟化，相反，每个容器与底层主机共享相同的硬件和 Linux 内核。尽管主机和硬件是共享的，Linux 容器允许运行和主机不同的 Linux 分发版本，只要求该版本与底层内核兼容。

例如，如果主机运行 Ubuntu，可以在容器中运行 CentOS。这是一项强大的功能，可以自由地为应用程序运行最好的 Linux 分发版本。

容器利用了多种 Linux 技术，但是 Linux 控制组和命名空间提供了容器的大部分好处，如资源占用率限制和隔离。

14.2.1　控制组

Linux 控制组（cgroup）明确地告诉内核，必须为进程分配多少 CPU 和 RAM 等资源。因为可以将相关的进程添加到一个组，CPU 调度器更容易决定如何为组成员分配资源。cgroup 的使用是必要的，因为容器和虚拟机不同，虚拟机有控制所有资源、实施合理资源分配的虚

拟化管理器，而容器对操作系统来说与常规的进程相似，cgroup 试图公平地分配资源。

14.2.2 命名空间

Linux 容器提供的隔离是由 Linux 内核的命名空间特性提供的。目前有 6 种命名空间：

- IPC（进程间通信）
- Mount（文件系统和挂载点）
- Network（联网）
- PID（进程）
- User（用户 ID）
- UTS（主机名）

每个命名空间提供应用程序之间的边界。在一个或者多个 Linux 命名空间内运行时，就称应用程序运行于一个容器之中。

Linux 命名空间的主要目的是提供进程隔离，你可以利用这种隔离构建和部署自足的应用程序集。每个应用程序集有自己的文件系统、主机名和网络栈。

应用程序和进程之间的交互有很多种方式，为了完全将一个进程与其他进程隔离，你需要在每个交互点创建边界。对于应用程序容器，主要需要处理挂载、UTS 和网络命名空间。其他命名空间专注于安全性，对运行不受信任的应用程序必不可少。

目前，我们已经描述了完全信任的应用程序或者进程的隔离。虽然可以利用其他 Linux 命名空间提供更好的隔离，但是 Liunx 容器自身不能提供 VMware 全部虚拟化技术所能达到的安全性级别。

挂载点（Mount）命名空间

挂载点命名空间提供两个运行进程之间的全面文件系统隔离，避免了依赖性冲突。本质上，每个应用程序可以看到不与主机操作系统共享的独立文件系统，每个应用程序可以在无冲突的情况下自由安装所依赖的模块。

使用挂载点命名空间与 chroots 相结合时，可以创建适合于运行应用程序的全隔离 Linux 安装。

UTS 命名空间

UTS（或者主机名）命名空间提供为每个应用程序实例指定唯一主机名的能力。

网络（Network）命名空间

网络命名空间为每个容器提供自己的网络栈，以消除两个运行应用之间的端口冲突。这意味着，每个应用程序可以指定自己的网络地址，使每个程序可以绑定相同的 TCP 端口而不会冲突。

查看内核文档，可以了解更多有关各个命名空间的信息。

14.2.3 容器管理

我们需要回答的下一个问题是如何创建 Linux 容器。这有一些困难，因为 Linux 命名空间的主接口是通过系统调用的。好消息是，可以使用 LXC 等用户空间工具，自动化容器的部署。

另一个消息则不那么好：即使使用 LXC，容器管理也需要处理其他一些难题：

- 在主机系统上运行容器 OS 的先决条件。
- 可能需要为资源控制而进行编辑的配置文件。
- 在不同分发版本或者相同分发版本家族内部，LXC 的实现可能不同。

14.3 使用 Docker

Docker 是一个客户端和服务器应用，提供构建和共享 Linux 容器的标准格式，简化 Linux 容器的使用。Docker 在 LXC、cgroup、命名空间和其他开源 Linux 项目的贡献基础上构建，用于比现有用户空间工具更进一步地简化容器部署。

从较高的级别上看，Docker 协助自动化 Linux 容器的创建，包括 Linux 内核提供的所有级别隔离。Docker 不仅能够简化容器化过程，还提供了简化不同主机之间容器共享的一种磁盘格式。

在某种意义上，Docker 将应用程序变成自给自足的可移植单位，构建一次即可在任何地方运行。我们在前面是不是已经听到了这种说法？

14.3.1 安装 Docker

Docker 解决方案由 Docker 守护进程和 Docker 客户端组成。Docker 守护进程当前只能在 x86_64 Linux 分发版本上使用，Microsoft 计划在未来的 Windows Server 版本上增加支持。Docker 客户端可以在 Linux、Windows 和 OS X 上运行，这使得从远程机器上执行命令非常方便。

在本书编写时，所有主流 Linux 分发版本都支持 Docker。

安装 Docker 的最常见方法是使用操作系统的软件包管理器。另一个选择是使用基本构建中包含 Docker 的 Linux 分发版本（如 CoreOS）。还可以使用 boot2docker，该工具利用一个非常小的 Linux 虚拟机（VM）运行容器。

如果在两个不同系统上运行 Docker 守护进程，两者一定要运行 Docker 的相同版本。

可以使用 docker version 命令检查安装的 Docker 版本：

```
$ docker version
Client version: 1.0.1
Client API version: 1.12
Go version (client): go1.2
Git commit (client): 990021a
Server version: 1.0.1
Server API version: 1.12
Go version (server): go1.2
Git commit (server): 990021a
```

14.3.2 Docker 守护进程

Docker 守护进程负责在单一主机上启动、停止和保存应用程序容器。它还提供远程应用程序编程接口（API），用于以编程方式控制容器。

14.3.3 Docker 客户端

Docker 客户端是一个命令行工具，可以通过 API 与 Docker 守护进程交互。Docker 客户端可以在 OS X 和 Linux 操作系统上以单一二进制程序的形式安装。为 Windows Servers 编译 Docker 客户端是可能的，但是这些步骤的介绍超出了本书的范围。

注意 在命令行输入 docker -h 可以看到所有可能操作的列表。

14.3.4 Docker 索引

Docker 索引是容器的存储库，类似于 Git 代码存储库。可以采用场内 Docker 索引，也可以使用 Docker 拥有的远程索引（http://hub.docker.com）。

14.3.5 运行 Docker 容器

研究创建 Docker 容器的方法之前，我们先使用 Docker 索引上预建的一个 Docker 映像，看看 Docker 的入门多么简单。我们将使用 docker run 命令和程序清单 14-1 中的语法，清单中还包含了输出示例。

程序清单 14-1 Docker 容器部署

```
$ docker run -d -p 6379:6379 --name redis redis
Unable to find image 'redis' locally
Pulling repository redis
dc10c1051e26: Download complete
511136ea3c5a: Download complete
1e8abad02296: Download complete
f106b5d7508a: Download complete
```

```
842b5a724d2d: Download complete
c48940f9fc45: Download complete
013e99580bf7: Download complete
076c6da2d5e4: Download complete
b7db07b1768c: Download complete
5e22b7a89800: Download complete
4b10ba128f08: Download complete
fd9a01509f91: Download complete
6dc9daf74f25906ea4f71ac64ef5fd92019acaf14434b1257d76b207e69855f8
```

-d 标志在容器创建之后断开，使其在后台运行。

-p 标志允许将容器中暴露的端口映射到命令行中指定的端口。在例子中，外部和内部端口映射到同一端口。

-name 标志允许用唯一的名称启动容器。在命名容器之后，提供想要运行的容器名称。在示例中，我们将运行 redis 容器映像。

在输出中可以看到，Docker 无法在本地找到 redis 映像。所以，Docker 继续下载该映像，每次一层。这包括 redis 映像所需的基本映像，默认情况下，Docker 将尝试从 Docker 公司维护的索引中下载映像。

层次代表增量构建容器映像的一种简易手段，加速了统一容器未来更新版本的下载。

输出中的最后一行是容器 ID。这个 ID 是我们收集容器其他相关信息的途径。因为我们用 -name 标志启动容器，所以也可以用名称代替容器 ID。

14.3.6　列出运行的容器

Docker 使容器的启动变得十分简单，以至于很容易失去指定系统上所运行容器的踪迹。使用 docker ps 命令，可以查看系统上运行的容器：

```
$ docker ps
CONTAINER ID   IMAGE          PORTS                    NAMES
66df5215853d   redis:latest   0.0.0.0:6379->6379/tcp   redis
```

从输出中可以看到，redis 容器从 redis Docker 映像，以容器 ID66df5215853d 启动。docker ps 输出还显示容器的端口映射。在 redis 容器中，主机上的端口 6379 映射到容器的端口 6379。

14.3.7　连接到运行的容器

如果在 Docker 的相同机器上安装 redis，可以使用如下语法连接到新的数据库实例：

```
$ redis-cli -h localhost -p 6379
```

如何获得这些连接参数？在 14.3.6 节中，我们有意地将容器中的应用程序端口（6379）映射到 Docker 主机的同一端口号。剩下的工作就是在 redis 客户端中指定所要连接的主机

（localhost）和端口号。

如果没有在本地安装 redis CLI，也没有任何问题！可以使用另一个容器连接到 redis 容器。

首先在 Docker 主机上创建第二个容器，我将使用简单的 Ubuntu 容器：

```
$ docker run -it --link redis:redis --name redis-client ubuntu /bin/bash
```

docker run 中有一些前面未讨论的新标志和选项。

-i 标志告诉 Docker，我们打算直接与容器交互，而不是立即断开。

-t 标志为我们提供一个 tty，以运行命令。

--link 标志带我们进入服务发现的领域，因为它通知新容器有关 redis 容器的重要特性。--link 标志将自动在新容器中创建对应于 redis IP 地址和应用程序端口号等信息的环境变量。程序清单 14-2 展示了一个示例。

最后，在容器名（ubuntu）之后，为新容器提供了一条运行命令（/bin/bash）。这样，当交互式容器启动时，我们自动进入 Bash Linux shell。

程序清单 14-2 Docker--link 环境变量

```
REDIS_ENV_REDIS_DOWNLOAD_URL=http://download.redis.io/releases/
    redis-2.8.19.tar.gz
REDIS_PORT_6379_TCP_PROTO=tcp
REDIS_ENV_REDIS_DOWNLOAD_SHA1=3e362f4770ac2fdbdce58a5aa951c1967e0facc8
REDIS_NAME=/redis-client/redis
REDIS_PORT_6379_TCP_ADDR=172.17.0.2
REDIS_PORT_6379_TCP_PORT=6379
REDIS_PORT_6379_TCP=tcp://172.17.0.2:6379
REDIS_PORT=tcp://172.17.0.2:6379
REDIS_ENV_REDIS_VERSION=2.8.19
```

执行 docker run 命令之后，我们将处于随机生成的主机名的命令提示符下：

```
root@fe86ff399421:/#
```

接下来，可以用如下命令安装 redis-cli：

```
# apt-get update
# apt-get install -y redis-tools
```

最后，用 Docker 的链接功能生成的环境变量连接到 redis 容器：

```
# redis-cli -h $REDIS_PORT_6379_TCP_ADDR -p $REDIS_PORT_6379_TCP_PORT
```

现在，可以访问 redis 数据库，并用几条简单命令确认：

```
172.17.0.2:6379> set devops rocks
OK
172.17.0.2:6379> get devops
"rocks"
```

14.3.8　构建和分发 Docker 容器

你可能觉得奇怪，Docker 社区中有许多出色的基本容器，为什么还要构建自己的容器。是的，当你连接到一个容器并运行命令（就像前面的 redis-client 示例）时，每次创建容器的实例，都必须完成这些更改。

如果我们可以分发自己的容器，预先准备所需的软件包和设置，不是很好吗？定制将成为基本映像之上的附加层次，用于新生成的容器。

Docker 容器格式由容器的文件系统表示和一些统称为 Docker 映像的元数据组成。Docker 映像包含如下项目：

- 一个压缩文件系统
- 一个元数据文件
- 容器配置文件，亦称 Dockerfile

我们将较为深入地讨论 Dockerfile，因为它是构建和自定义容器的手段。在继续本章的其余讨论之前，在 Docker Hub（http://hub.docker.com）上开立一个账户，为简单起见，可以使用和 GitHub 账户相同的用户名。本章后面将对 Docker Hub 略加讨论。

关于 Docker 容器使用哪种文件系统（aufs 和 btrfs 对比）更好的讨论超出了本书的范围。但是，Docker 容器默认保存在 Docker 主机的 /var/lib/docker 上。

14.3.9　Dockerfile

Dockerfile 是一个文本文档，包含一组用于构建容器的命令。这和第 3 章中介绍的，以 Vagrant 自动化测试环境构建的 Vagrantfile 是类似的概念。

程序清单 14-3 是一个用于构建 Nginx Web 服务器的 Dockerfile 示例。

程序清单 14-3　Nginx 容器的 Dockerfile

```
# Ubuntu Trusty release selected for the web server
FROM ubuntu:14.04
MAINTAINER Trevor Roberts Jr <vmtrooper@gmail.com>

# Install the add-apt-repository utility
RUN apt-get -qq update && apt-get install -qq -y software-properties-common

# Add the nginx stable repository
RUN add-apt-repository -y ppa:nginx/stable && apt-get -qq -y update

# Install nginx
RUN apt-get install -qq -y nginx
```

正像程序清单 14-3 中的第一行，我们用 # 符号添加了有用的注释，任何修改 Dockerfile 的人都能理解容器构成的思想过程。

接下来，FROM 语句告诉我们所使用的基本容器。FROM 命令中容器标识符的格式为 repository:tag。在我们的例子中，容器来自于 Ubuntu 存储库，标记为 14.04。有些存储库中有多个带有标记的容器，包括标记为“latest”（最新）的容器。如果没有指定标记，将自动使用标记为“latest”的容器映像。

Docker 搜索 FROM 命令指定的容器时，首先在本地搜索映像，如果在本地没有找到，Docker 守护进程搜索 http://hub.docker.com 等远程存储库。

接下来，MAINTAINER 命令识别维护容器的人。一定要用自己的联络信息更新这一部分。

最后，RUN 命令向 Docker 守护进程提供如何构建容器的命令。

注意，例中使用 && 符号组合同一行中的命令。可以使用 && 符号使关联命令成为同一 Docker 构建步骤的一部分。这些关联命令成为同一 Docker 层次的一部分，可以加速其他人下载容器的过程。

例如，第一条 RUN 命令执行必备的初始化 apt 存储库更新，然后安装 software-properties-common 软件包。这个包中包含了下一条 RUN 命令使用的 add-apt-repository 实用工具。

因为这两条命令的执行有一定关联，它们可以成为同一构建步骤的一部分。记住，Docker 容器构建过程是自动化的，没有用户输入的机会。就像在 shell 脚本中使用这些命令一样，验证命令执行中包含了自动化输入请求以及可选抑制输出的正确命令行选项很重要。

现在，我们的 Dockerfile 必须进行处理，以提供可工作的容器映像，这项工作由 docker build 命令完成。下面的语法说明如何构建刚刚创建的 Dockerfile：

```
$ docker build -t="your_docker_hub_username/nginx:v1" .
```

build 命令的 -t 选项可以指定存储库名，也可以选择应用到该容器的标记。存储库名称由 Docker Hub 用户名和为容器提供的名称（例如，nginx）组成。例子中的标记是“v1”，标记是可选的，如果没有包含标记，这次构建的映像在 Docker Hub 中将被标记为“latest”。

最后，提供 Dockerfile 的路径。如果从保存 Dockerfile 的同一目录下执行容器构建，可以使用句点（.）告诉守护进程，在当前工作目录下查找。

14.3.10 Docker Hub

拥有工作容器之后，可以研究将其推送到某个索引的方法。关于使用的索引，有几种选择：

- 已经在内部部署的索引
- 第三方主机托管提供商提供的索引
- Docker 公司运营的 Docker 索引（http://hub.docker.com），称作 Docker Hub

如果构建程序清单 14-3 中的容器示例，可以使用如下命令将其推送到 Docker Hub 索引：

```
docker push your_docker_hub_username/nginx:v1
```

push 命令告诉 Docker 守护进程，传输容器独特的构建步骤，注意我所说的“独特”。如果从现有容器中构建一个容器，推送过程会在连接到 Docker Hub 时发现，只保存引入的更改。Docker Hub 支持公共和私有的容器存储库。

14.3.11　Docker 与虚拟机的对比

容器是分配给给定任务的计算资源离散单元，与 VM 类似。这是否意味着，它们可以消除 VM 的使用？完全不是这样。VM 有安全性隔离和基础设施部署速度等容器无法解决的使用场景。

谈到隔离，VM 中的进程不会像容器那样影响物理主机上的其他进程。这并不是说容器本身不安全，但是，如果容器中的应用程序需要根权限或者其他可能带来安全问题的配置，最好是将它和相关的容器隔离到 VM 或者 VM 组。

至于基础设施部署，尽管有 Razor 和 Cobbler 等出色技术能够快速启动裸机服务器，VM 的部署仍然比裸机服务器的部署快几个数量级。如果物理基础设施有足够的空闲容量，可以部署 VM 以提供工作负载的必要隔离，然后在这些 VM 中用容器部署应用程序。

14.3.12　Docker 与配置管理的对比

Docker 和配置管理技术一样加速应用程序部署。这是否意味着，我们要摆脱 Puppet 清单或者 Chef 食谱？答案是“不”。

配置管理技术可以和容器相结合，用于应用程序部署。用例之一是将配置管理脚本的执行加入 Dockerfile。这样做是有意义的，因为你可能已经投入时间确保配置管理命令文件中有合适的相互依赖性，不必在 Dockerfile 中推倒重来，而可以从 Dockerfile 中执行配置管理脚本。

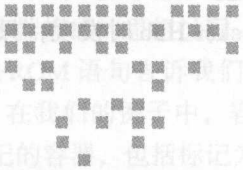
14.4　小结

本章中，我们定义了应用程序，并研究了应用程序的构建和部署。我们还了解了传统操作系统的局限性，以及给应用程序部署带来的复杂性。最后，我们讨论了如何利用应用容器降低应用程序构建和部署的复杂度。第 15 章介绍使用在 VM 技术基础上运行的分布式 Linux OS，规模化利用 Linux 应用容器规划和部署应用的细节。

参考文献

[1] Docker documentation: http://docs.docker.com

[2] Docker CLI for Windows: http://azure.microsoft.com/blog/2014/11/18/docker-cli-for-windows-clients/



Chapter 15 第 15 章

大规模运行 Docker 容器

第 14 章解释了在 Docker 中容器实际上是奇特的 Linux 进程，自带 Docker 格式的所有依赖性。在单一系统上，进程通常由 init 系统和 Linux 内核管理。但是，我们如何管理多台主机上的容器？这需要一种编排解决方案，不仅部署容器，还必须处理其他后勤问题，如容器间通信、容器状态管理（例如，运行、停止、从故障中恢复）等。本章专注于在多台主机上实现大规模容器管理的技术。

本章包含如下主题：

- 容器编排
- Kubernetes
- Kubernetes 部署
- 用 Docker 实现平台即服务

15.1 容器编排

从 Docker 项目公布起，容器领域发生了一些有趣的事情。我们已经发现社区中对使用容器进行应用部署有浓厚的兴趣。而且，我们所称的容器优化操作系统越来越流行，这种操作系统只自带必要的组件和配置设置，可以立即运行容器，通常意味着预先安装了 Docker。

Linux 分发版本 CoreOS 已经成为最著名的容器操作系统，原因很充分。CoreOS 没有携带 apt 或者 yum 等包管理器，所有软件必须用 Docker 部署为 Linux 容器。

不过，为了真正大规模地运行容器，需要解决另一些难题，如布置（亲和或者反亲和）、可用性、联网等。有许多工具可以用于这些问题，但是它们往往有一个共同的特征：全都专

注于将特定的容器固定在特定主机上。确实可以用这种方法部署许多容器，但是无法发挥容器所具备的所有潜力。

现在 Docker 已经来到，我们发现自己正在创建比以往更多的进程，必须将其部署到整个群集上。群集的规模增长归功于虚拟化和在任何地方部署任何系统的能力。

在单一主机上，init 系统为我们提供了停止、启动和重启进程的能力。正常启动运行之后，进程经过调度，获得硬件和内存的访问权。这是系统管理员 30 多年来管理进程的方式。今天，我们所需要的是以同样的方式调度群集中不同机器上的“超级”进程（即 Linux 容器）的能力，而且还要保持和单一主机相同的易用性，这是 Kubernetes 的切入点。

15.2　Kubernetes

Kubernetes 是来自 Google 的容器管理系统，它提供了独特的工作流，以管理多台机器上的容器。Kubernetes 引入了“豆荚”（pod）的概念，代表一组在相同主机上运行、共享网络和文件系统的相关容器。豆荚被作为单一逻辑服务部署，这个概念很适合于流行的一种模式：每个容器运行单一服务。Kubernetes 使在一台机器或者整个群集上运行多个豆荚变得简单，从而得到更好的资源利用率和高可用性。Kubernetes 还主动监控豆荚的健康，确保它们始终在群集内运行。

Kubernetes 利用 etcd 分布式键 - 值存储（CoreOS 启动和维护的一个项目）实现这种高级的群集范围编排。etcd 负责整个群集内 Kubernetes 所使用数据的存储和复制，采用 Raft 一致性算法从硬件故障和网络分区中恢复。在专为这类目的构建的 CoreOS 操作系统上部署时，Kubernetes 和 etcd 的组合威力更加强大。

Kubernetes 工作流

看待 Kubernetes 的简单方法之一是，将其视为以“数据中心即计算机”的思路为基础构建的项目，Kubernetes 提供了应用程序编程接口（API）管理所有事务。

Kubernetes 提供的工作流允许我们将群集作为整体来考虑，而较少考虑单独的机器。在 Kubernetes 模型中，我们不将容器部署到特定的主机，相反，我们描述所要运行的容器，Kubernetes 计算出运行的方式和位置。这种声明性的基础设施管理风格为大规模部署和自修复基础设施打开了大门。欢迎来到未来世界。

但是，这一切是如何做到的？

Kubernetes 豆荚对于从主机托管服务中得益的应用程序（如缓存服务器或者日志管理器）很合适。豆荚在 Kubernetes 群集中通过规格文件创建，这些规格文件被推送到 Kubernetes API。Kubernetes 调度程序将选择一个群集节点部署豆荚。如果规格文件表明豆荚应该复制，Kubernetes 控制器管理程序将验证每个应用程序规格所需部署的豆荚数量。

每个群集节点上运行两个服务：Kubelet 和 Kubernetes Proxy（代理）。Kubelet 接收 Kubernetes 指令，通过 Docker API 启动相关容器，负责保持调度的容器正常运行，直到豆荚被删除或者调度到另一台机器。

Kubernetes Proxy 服务器负责将服务的 TCP 端口映射到应用程序豆荚。思路是服务消费者可以在服务指定的端口上联系 Kubernetes 群集中的任何机器，请求被路由到正确的豆荚，提供对应用程序的访问。这种通信的实现是因为每个豆荚有自己的 IP 地址，Kubernetes 使用 etcd 跟踪豆荚 IP，能够相应地将应用程序访问请求路由到正确的豆荚和主机。

Kubernetes 根据标签提供服务发现的一种形式，使工作更加轻松。每组豆荚可以有任意数量的标签，用于定位它们。例如，要定位生产环境中运行的 redis 服务，可以使用如下标签执行标签查询：

```
service=redis,environment=production
```

15.3 Kubernetes 部署

我们的 Kubernetes 环境由一个 3 节点的 CoreOS 群集组成。下面的软件组件组成了演示环境：

- CoreOS Linux
- etcd
- flannel（用于为豆荚提供 IP 的简单网络覆盖技术）
- fleet（用于在 CoreOS 群集上部署应用程序的实用工具）
- Kubernetes
- Kubernetes Register（kube-register）（简化在 Kubernetes API 服务器上注册 Kubelet 的开源项目）
- etcdctl、fleetctl 和 kubectl（分别管理 etcd、fleet 和 Kubernetes 的 CLI 工具）

15.3.1 CoreOS 和 Kubernetes 群集管理工具

部署群集节点之前，我们首先要获得帮助管理环境的 CLI 工具。可以从 GitHub 的下列位置获取二进制程序：

- etcdctl—https://github.com/coreos/etcd/releases
- fleetctl—https://github.com/coreos/fleet/releases
- kubectl—https://github.com/GoogleCloudPlatform/kubernetes

kubectl 二进制程序实际上需要在使用之前构建。好消息是，Google 团队提供了一个用于构建的容器，使构建过程非常容易实现。唯一的先决条件是在本地安装 Docker（如果桌面系统是 Windows 或者 Mac OS X，则是 boot2docker）。

验证 docker 运行之后（例如，docker ps 成功执行），在本地克隆 Kubernetes 存储库，并

将目录更改为刚刚下载的“kubernetes”目录。接下来，运行如下命令：

```
$ ./build/run.sh hack/build-cross.sh
```

如果 Docker 在系统上正常工作，kubernetes 二进制程序的构建就会成功继续，在如下目录中将会找到用于你的操作系统的 kubectl 二进制程序：

```
kubernetes/_output/dockerized/bin
```

在本书编著时，fleetcle 的 Windows 二进制程序不存在。如果有 Windows 桌面，可以尝试从源代码构建二进制程序，或者使用 Vagrant（参见第 3 章）创建一个小的 Linux VM，以管理 Kubernetes 环境。

二进制程序应该复制到当前路径下的一个目录（例如，/usr/local/bin）。选择 etcd 服务器 IP 之后，我们将讨论这些二进制程序与 Kubernetes 群集通信所需环境变量的设置。

15.3.2　CoreOS 群集部署

运行 CoreOS 操作系统的 3 个 VM 可以用来自其网站的 CoreOSbuilt VMware 映像（推荐）或者人工安装 CoreOS 部署。3 个 VM 都必须连接到同一个 VMware vSphere 端口组或者 VMware Fusion/Workstation vmnet。

注意　连接这些 VM 的 vSphere 端口组或者 Fusion/Workstation vmnet 必须运行动态主机配置协议（DHCP），实验室的设置才能正常工作。建议不要用 DHCP 管理整个网段，以便于为 etcd 服务器分配静态 IP。例如，如果网络名为 192.168.10.0/24，可以考虑将 DHCP 服务管理地址放在较高的 IP 范围，如 192.168.10.100 -192.168.10.254。

接下来，我们创建云配置文件，规定 CoreOS 服务器引导时的配置。幸运的是，这些文件已经创建好，保存在本环境 GitHub 存储库（https://github.com/DevOpsForVMware Administrators/kubernetes-fleet-tutorial）的 /configs 目录下，只需要更新文件中从属于你的环境的设置。必要的修改在后面描述，我们在程序清单 15-1 和 15-2 中分别展示了用于 etcd 和 Kubernetes 的配置文件样板。

程序清单 15-1　etcd 服务器配置文件

```
#cloud-config

hostname: etcd
coreos:
  fleet:
    etcd_servers: http://127.0.0.1:4001
    metadata: role=etcd
  etcd:
    name: etcd
```

```
    addr: 172.16.171.10:4001
    bind-addr: 0.0.0.0
    peer-addr: 172.16.171.10:7001
    cluster-active-size: 1
    snapshot: true
  units:
    - name: static.network
      command: start
      content: |
        [Match]
        Name=enp0s17

        [Network]
        Address=172.16.171.10/24
        DNS=172.16.171.2
        Gateway=172.16.171.2
    - name: etcd.service
      command: start
    - name: fleet.service
      command: start
    - name: docker.service
      mask: true
  update:
    group: alpha
    reboot-strategy: off
ssh_authorized_keys:
    - ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCXJo3tfAdsvBZp/l6ChpYbKYtTsK-
cAL32VhV7KGjYqUPY9LJhRPlAAfAh6bU1hBf9wLTCWVOAoWwrQg09raJadfCbDsNohMy2t-
P37vGxyCdd/55uejWIkZEVeMJgRiQC7WIL1GMNWXf8dwWDvk4Xx0MbRBUpb5LcwPFnsCyQEV-
cyi0NQT+1odNnjt74rYCBPbVrM1Owski3zizt+JWqSNAPCYaj+Yvy3OulUmkhZIv8OsEeL-
4BACiZNQDAJFhVw7DW4LZicw+GNzavHD7M0Q5UQdIwx43h0Oiw6AEKtHD16TuYgAVs2ult-
K95Xxg3atZO/WCcgBt2ObFRNOaPAjf3p
```

程序清单 15-2 Kubernetes 服务器配置文件

```
#cloud-config

coreos:
  fleet:
    etcd_servers: http://172.16.171.10:4001
    metadata: role=kubernetes
  units:
    - name: etcd.service
      mask: true
    - name: fleet.service
      command: start
```

```
    - name: flannel.service
      command: start
      content: |
        [Unit]
        After=network-online.target
        Wants=network-online.target
        Description=flannel is an etcd backed overlay network for contain-
ers

        [Service]
        ExecStartPre=-/usr/bin/mkdir -p /opt/bin
        ExecStartPre=/usr/bin/wget -N -P /opt/bin http://storage.googlea-
pis.com/flannel/flanneld
        ExecStartPre=/usr/bin/chmod +x /opt/bin/flanneld
        ExecStart=/opt/bin/flanneld -etcd-endpoint
http://172.16.171.10:4001
    - name: docker.service
      command: start
      content: |
        [Unit]
        After=flannel.service
        Wants=flannel.service
        Description=Docker Application Container Engine
        Documentation=http://docs.docker.io

        [Service]
        EnvironmentFile=/run/flannel/subnet.env
        ExecStartPre=/bin/mount --make-rprivate /
        ExecStart=/usr/bin/docker -d --bip=${FLANNEL_SUBNET} --mtu=${FLAN-
NEL_MTU} -s=btrfs -H fd://

        [Install]
        WantedBy=multi-user.target
    - name: setup-network-environment.service
      command: start
      content: |
        [Unit]
        Description=Setup Network Environment
        Documentation=https://github.com/kelseyhightower/setup-network-
environment
        Requires=network-online.target
        After=network-online.target
        [Service]
        ExecStartPre=-/usr/bin/mkdir -p /opt/bin
        ExecStartPre=/usr/bin/wget -N -P /opt/bin http://storage.
googleapis.com/snenv/setup-network-environment
```

```
        ExecStartPre=/usr/bin/chmod +x /opt/bin/setup-network-environment
        ExecStart=/opt/bin/setup-network-environment
        RemainAfterExit=yes
        Type=oneshot
  update:
    group: alpha
    reboot-strategy: off
ssh_authorized_keys:
    - ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDQgbpj/
u2FTiBBSqBElIrcYCSg9YP2V4VRH/ckV7qWxRBlHwCBGgJSoG+h-
63HG59Jlpe7vtCSlXcxNUbV997AsDMPKGRiww6ngWqyr8GJT+3/
LNSRGAKFOjyhfU4VUYRS9fQNdr0FTfWWDKKgTTyQVdrom+jFw5MBERwBS2u2Srmgw6NWC9BEZn/
j7SWfI3HxdmfWvXWbZO6Ujv3mq2/tVw/2WexspfQ7Nnkcysrvgj YfvzlUNrwRrqQhaUL-
8N5+2+C09PzyCVCl7WvVJy+PoBOHeOlVxohuLsLQvQEdTozdtiBCW85W5lMzHSjiYNRTAf+Qw-
zuJhD3Lrh3p/MSB9B
```

在程序清单 15-1 中，需要更新如下值以反映实验室环境：

- etcd：段
  - addr：用来自自有 VM 网络的静态 IP 代替 IP 地址，不要改变端口号（4001）。
  - peer-addr：使用和上面相同的 IP，同样，不要更改端口号（7001）。
- units：段
  - [Match] Name=：这指的是 VM 网络接口的名称。一旦 VM 引导，就可以看到这个值。图 15-1 中是一个例子，VM 网络接口名称 enp0s17 出现在登录提示符之上。
  - [Network] Address、DNS 和 Gateway：这些值应该与选择的 IP 和端口组 /vmnet 的对应设置相符。
- ssh_authorized_keys：用自己的安全外壳（SSH）公钥值代替该值。注意 YAML 语法，确保条目前的连字号，正如样板文件中那样。

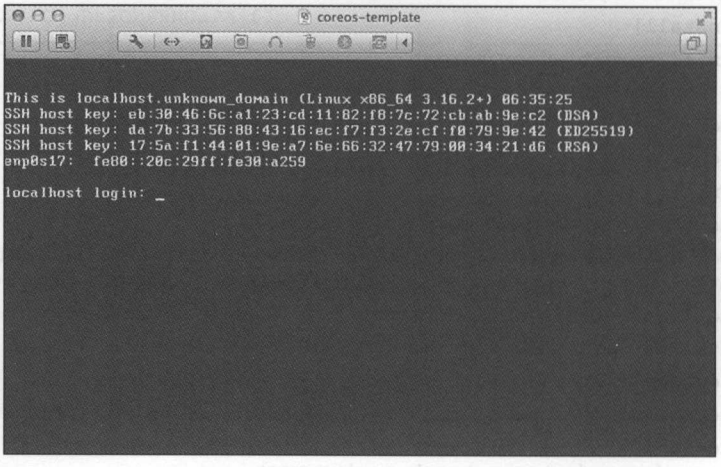


图 15-1 CoreOS VM 引导提示

在程序清单 15-2 中，需要更新如下值以反映实验室环境：

- fleet：段
- -etcd_servers：用 etcd 服务器配置文件所用的静态 IP 地址代替该地址。
- units：段
- -flannel.service：更改 etcd-endpoint，匹配 etcd 服务器的 IP 地址。
- ssh_authorized_keys：用自己的 SSH 公钥值代替该值。注意 YAML 语法，确保条目前的连字号，正如样板文件中那样。

注意　etcd 服务器是唯一需要静态 IP 的服务器。两个 Kubernetes 节点和以后添加到 Kubernetes 群集中的服务器将使用 DHCP 获取 IP。

YAML 文件必须按照如下惯例写入 ISO 文件（每个 YAML 一个 ISO）：

- 卷标必须为 config-2。
- YAML 必须更名为 user_data，放在 hierarchy/openstack/latest/ 文件夹中，openstack 目录在 ISO 文件的根目录中。
- 在桌面上挂载 ISO，验证 ISO 正确创建，卷标应该显示为 config-2。ISO 的根目录应该包含 openstack 目录，user_data 文件保存在 latest 子目录中。

15.3.3　etcd 服务器配置

现在，你应该有两个 ISO 文件：一个用于 etcd 服务器，另一个用于 Kubernetes 群集节点。为了简单起见，我们将把 etcd 服务器所用的 ISO 文件命名为 coreos-master.iso，用于 Kubernetes 群集节点的 ISO 文件命名为 kubernetes-node.iso，建议遵循类似的命名标准。

将 coreos-master.iso 连接到将作为 etcd 服务器的 CoreOS VM，将 kubernetes-node.iso 连接到其他两个 VM。在此时只重新启动 etcd 服务器 VM。（我们将在稍后启动剩下的两个服务器。）

在前面谈及 CLI 实用工具时，我们曾经提到，需要设置环境变量，这些实用工具才能正常工作。这两个环境变量是 ETCDCTL_PEERS 和 FLEETCTL_ENDPOINT。你将使用 etcd 的服务器和端口 4001，如：

```
$ export ETCDCTL_PEERS=http://192.168.12.10:4001
$ export FLEETCTL_ENDPOINT=http://192.168.12.10:4001
```

一定要用自己的 etcd 服务器 IP 地址替换示例中的值，如果需要重新打开终端窗口，可以将这些语句放在容易找到的文件中。

etcd 服务器重新启动完成后，可以使用如下命令验证 etcdctl 和 fleetctl 正常工作：

```
$ etcdctl ls /
$ fleetctl list-machines
```

如果两条命令都没有出错，etcd 服务器就已经正常运行。

你可能觉得奇怪，etcd 服务器对群集这么重要，为什么只设置一个？这种担忧很有道理。在生产环境中，应该部署一个包含奇数台主机的 etcd 群集，而不是使用单个 etcd 服务器。这样能够确保重要资源的高可用性。但是，在我们的演示中，单个 etcd 服务器足够了。

15.3.4 Flannel 网络覆盖

如前所述，每个豆荚都有自己的 IP 地址。标准的管理是利用私有 IP 编址，这可以用多种途径实现（VM 上的第二个网卡 [NIC]，为单一 NIC 分配多个 IP 地址等）。出于演示目的，我们利用 CoreOS 的 flannel 项目创建简单的覆盖网络，将其映射信息保存在 etcd。下面的命令为 flannel 分配 /16 地址空间：

```
$ etcdctl mk /coreos.com/network/config '{"Network":"10.0.0.0/16"}'
```

我们使用默认的 flannel 设置，为 Kubernetes 群集中的每台主机分配一个 /24 IP 地址空间。etcd 将包含 Kubernetes 引用的映射，以确定哪台物理主机拥有特定的私有地址空间。例如，在 Kubernetes 设置好并正常运行时，下面的 etcdctl 查询将看到覆盖网络范围与主机的映射：

```
$ etcdctl get /coreos.com/network/subnets/10.0.76.0-24
{"PublicIP":"172.16.171.134"}
```

在这个样板输出中，PublicIP 引用的是 Kubernetes 群集节点的 IP 地址，10.0.76.0-24 是分配给该服务器的 IP 地址空间。

15.3.5 Kubernetes 群集节点

我们将 kubernetes-node.iso 文件连接到余下的两个 CoreOS VM 并重启它们进行验证。监控 VM 控制台窗口，验证 CoreOS 自定义何时完成。在 VM 控制台上看到登录提示符时，服务器已经就绪。也可以使用 fleetctl 验证节点状态：

```
$ fleetctl list-machines
MACHINE         IP              METADATA
19de047d...     172.16.171.133  role=kubernetes
775e2223...     172.16.171.134  role=kubernetes
98786c6b...     172.16.171.10   role=etcd
```

你应该看到列出了 3 台机器，其中之一是 etcd 角色，其余两台服务器是 kubernetes 角色。这些元数据将用于识别哪些节点应该运行 Kubernetes 服务。

在 CoreOS 中，当你部署应用时，使用 systemd 单元文件，该文件包含应用程序部署指令、需要部署的容器、部署应用的主机类型、容器布置中是否应该有亲和性等。本例存储库中的 units 目录包含设置 Kubernetes 群集所需的所有部署指令。程序清单 15-3 和 15-4 展示了两个单元文件的细节。

程序清单 15-3　kube-scheduler.service

```
[Unit]
Description=Kubernetes Scheduler Server
Documentation=https://github.com/GoogleCloudPlatform/kubernetes

[Service]
ExecStartPre=/usr/bin/wget -N -P /opt/bin http://storage.googleapis.com/
kubernetes/scheduler
ExecStartPre=/usr/bin/chmod +x /opt/bin/scheduler
ExecStart=/opt/bin/scheduler --master=127.0.0.1:8080
Restart=always
RestartSec=10

[X-Fleet]
MachineOf=kube-apiserver.service
MachineMetadata=role=kubernetes
```

程序清单 15-4　kube-kubelet.service

```
[Unit]
Description=Kubernetes Kubelet
Documentation=https://github.com/GoogleCloudPlatform/kubernetes
Requires=setup-network-environment.service
After=setup-network-environment.service

[Service]
EnvironmentFile=/etc/network-environment
ExecStartPre=/usr/bin/wget -N -P /opt/bin http://storage.googleapis.com/
kubernetes/kubelet
ExecStartPre=/usr/bin/chmod +x /opt/bin/kubelet
ExecStart=/opt/bin/kubelet \
--address=0.0.0.0 \
--port=10250 \
--hostname_override=${DEFAULT_IPV4} \
--etcd_servers=http://192.168.12.10:4001 \
--logtostderr=true
Restart=always
RestartSec=10

[X-Fleet]
Global=true
MachineMetadata=role=kubernetes
```

在单元文件中，Unit 段提供正在运行的应用程序的相关基本信息，以及和群集中其他服务相对的启动顺序。Service 段指定将要运行的二进制程序和所有启动选项。最后，

X-Fleet 段提供了应用程序应该在 CoreOS 群集中的哪个位置运行的细节。比较程序清单 15-3 和 15-4 中的 X-Fleet 段，会发现它们都设置为仅在有 Kubernetes 角色的 CoreOS 服务器上运行。

但是，有两处差异：

- Kubernetes Scheduler 服务是 Kubernetes 控制栈的一部分。所以，我们使用 MachineOf 设置指定与 Kubernetes API 服务的亲和性。
- Kubelet 服务必须在所有 Kubernetes 群集节点上运行。所以，我们将 Global 设置为 true，确保这一点。

15.3.6 Kubernetes 服务部署

我们需要修改几个文件，指向你的 etcd 服务器 IP 地址，具体地说是 kube-apiserver.service、kube-kubelet.service 和 kube-proxy.service 中的 etcd_servers 值。一旦文件更新，我们将使用 fleetctl 部署单元文件：

```
$ fleetctl start kube-proxy.service
$ fleetctl start kube-kubelet.service
$ fleetctl start kube-apiserver.service
$ fleetctl start kube-scheduler.service
$ fleetctl start kube-controller-manager.service
$ fleetctl start kube-register.service
```

使用 fleetctl list-units 命令验证核心 Kubernetes 正常设置运行：

```
$ fleetctl list-units
UNIT                            MACHINE                         ACTIVE  SUB
kube-apiserver.service          1f4a2e45.../172.16.171.128      active  running
kube-controller-manager.service 1f4a2e45.../172.16.171.128      active  running
kube-kubelet.service            19de047d.../172.16.171.133      active  running
kube-kubelet.service            1f4a2e45.../172.16.171.128      active  running
kube-proxy.service              19de047d.../172.16.171.133      active  running
kube-proxy.service              1f4a2e45.../172.16.171.128      active  running
kube-register.service           1f4a2e45.../172.16.171.128      active  running
kube-scheduler.service          1f4a2e45.../172.16.171.128      active  running
```

前面，我们将 kube-register 组件与核心 Kubernetes 区分开。正常情况下，Kubernetes API 服务必须预先知道 Kubelet 主机的 IP 地址。CoreOS 的 Kelsey Hightower 贡献的 kube-register 服务自动将 Kubernetes 节点注册到 API 服务。这通过监控运行 Kubelet 服务的服务器 etcd，在新服务器上线时通知 API 服务来实现。

值得注意的是，尽管 Kubernetes 部署过程看起来很复杂，我们到目前为止所做的事情简化了更多 Kubernetes 群集节点的添加。如果想要增大群集的规模，部署更多的 CoreOS VM 并将 kubernetes-node.iso 节点连接到 CD-ROM，使用和群集其余机器相同的虚拟网络。在这些服务器引导时，它们将自动添加到 Kubernetes 群集，这归功于 kube-register 服务。

15.3.7　Kubernetes 工作负载部署

kubectl 实用工具利用 Kubernetes RESTful API 部署和管理 Kubernetes 群集上的工作负载。在使用它之前，需要设置环境变量，告诉 kubectl 哪一个群集节点运行 Kubernetes API 服务。(从 fleetctl list-units 命令获取 IP 地址。)

```
$ export KUBERNETES_MASTER="http://172.16.171.128:8080"
```

我们可以使用 kubectl 列出运行 Kubelet 的服务器和代理服务（Kubernetes 术语中称为“奴才”(minions)）的服务器，验证 Kubernetes 群集正常设置运行，命令如下。

```
$ kubectl get minions
NAME                LABELS              STATUS
172.16.171.128      <none>              Unknown
172.16.171.133      <none>              Unknown
```

多层应用程序部署

现在，我们已经验证 Kubernetes 群集正常运行，可以部署应用程序了。和前几章一样，我们将关注 MySQL 数据库、Apache Web 服务器和 PHP 应用服务器的部署，只是这次我们将这些应用作为 Docker 容器部署，使用 Kubernetes 管理服务的布置和可用性。

在前面关于容器的章节中，我们介绍了 Docker hub，将利用它来获取工作负载。MySQL 有一个标准容器，将用于数据库。我们将为 Web 和应用服务器组件使用自定义容器。部署应用程序时，我们将遵循每个豆荚一个应用的惯例，这可以简化故障检修和可用性设计。

Kubernetes 接受 JavaScript 对象标记法（JSON）格式的指令。JSON 是用于描述数据的普通文本格式，类似于 YAML，一般用于发送 / 读取 Web API 数据，是可扩展标记语言（XML）的流行替代品。每次应用程序部署将包含 2 个 JSON 文件：一个豆荚定义文件和一个服务定义文件。你可以在本环境 GitHub 存储库（https://github.com/DevOpsForVMwareAdministrators/kubernetes-fleet-tutorial）的 applications 目录下找到示例 JSON 文件。

豆荚定义

第一个指令文件告诉 Kubernetes 要部署哪一个应用程序，以及部署的方式。程序清单 15-5 展示了 MySQL 应用程序部署的 JSON 文件示例。

程序清单 15-5　MySQL 豆荚定义 JSON 文件（mysql-pod.json）

```
{
  "id": "mysql-server-id",
  "kind": "Pod",
  "apiVersion": "v1beta1",
  "desiredState": {
    "manifest": {
      "version": "v1beta1",
      "id": "mysql-server-id",
```

```
      "containers": [{
        "name": "database",
        "image": "dockerfile/mysql",
        "ports": [{
          "containerPort": 3306,
          "hostPort": 3306
        }]
      }]
    }
  },
  "labels": {
    "name": "mysql-server"
  }
}
```

让我们来观察豆荚定义文件中的一些要素。id 值是豆荚在 etcd 系统中的标识。kind 值代表我们打算部署的 Kubernetes 组件类型。desiredState 段包含容器规格，包括容器名称、容器应用程序使用的端口（containerPort），以及容器通信使用的主机端口。最后，labels 段是用户和其他容器标识使用 Kubernetes API 部署的应用程序的方法。

服务定义

第二个指令文件规定 Kubernetes 如何将流量导向可用的 MySQL 豆荚。程序清单 15-6 展示了用于数据库的服务文件。

程序清单 15-6 MySQL 服务定义 JSON 文件（mysql-service.json）

```
{
  "id": "mysqlserver",
  "kind": "Service",
  "apiVersion": "v1beta1",
  "port": 10000,
  "containerPort": 3306,
  "selector": {
    "name": "mysql-server"
  }
}
```

Kubernetes Proxy 使用 port 值向用户和其他豆荚公告容器服务。selector 是发送到该服务的流量将要转发的目标豆荚标签。记住，在程序清单 15-5 中，标签是我们最后设置的值。

转向 Web 和应用层，程序清单 15-7 和 15-8 分别展示了 Web 和应用程序容器的豆荚和服务定义文件。

程序清单 15-7　Web 和应用程序豆荚定义 JSON 文件（web-pod.json）

```
{
  "id": "web-server-id",
  "kind": "ReplicationController",
  "apiVersion": "v1beta1",
  "desiredState": {
    "replicas": 2,

    "replicaSelector": {"name": "web-server"},
    "podTemplate": {
      "desiredState": {
        "manifest": {
          "version": "v1beta1",
          "id": "web-server-id",
          "containers": [{
            "name": "web",
            "image": "vmtrooper/web-app",
            "ports": [{
              "containerPort": 80,
              "hostPort": 80}]
          }]
        }
      },
      "labels": {"name":"web-server"}
    }
  },
  "labels": {
    "name": "web-server"
  }
}
```

注意，我们在豆荚定义文件中使用稍有不同的结构。我们打算使用复制的 Web 和应用层，如果其中一个节点出现故障，可以在很少或者完全没有宕机时间的情况下保持可用性。

当我们想让 Kubernetes 控制器管理程序创建豆荚副本时，为 kind 值指定 Replication Controller 关键字。我们还指定副本的数量和被复制豆荚的标签。另一个新元素是 podTemplate 对象，用于定义控制器管理程序在每个服务器上部署豆荚时使用的豆荚特性。

程序清单 15-8　Web 和应用程序服务定义文件（web-service.json）

```
{
  "id": "webserver",
  "kind": "Service",
  "apiVersion": "v1beta1",
  "port": 10080,
```

```
    "containerPort": 80,
    "labels": {
         "name": "web-server"

    },
    "selector": {
      "name": "web-server"
    }
  }
```

复制的豆荚服务定义文件没有很多不同之处，但是我们在这里列出，以说明可以为服务指定标签，根据标签执行 Kubernetes 服务查询：

```
kubectl get services -l "name=web-server"
```

现在我们已经完成了定义文件，可以继续用如下的语法启动它们。(注意 create 之后的关键字。)

```
$ kubectl create -f mysql-pod.json
$ kubectl create -f mysql-service.json
$ kubectl create -f web-pod.json
$ kubectl create -f web-service.json
```

根据互联网连接的速度，每个豆荚的状态可能需要几分钟才能变成 Running (运行中)。可以使用如下命令验证豆荚状态：

```
$ kubectl get pods
$ kubectl get services
```

web-app 实例可以在任一 Kubernetes minion 上的端口 10080 访问，也可以使用 MySQL 客户端连接到任一 Kubernetes minion 上的端口 10000 管理数据库。(记住，服务连接被路由到正确的容器主机。) 而且，如果在连接 kubernetes-node.iso 文件的情况下启动另一个 CoreOS VM，它将自动变成 Kubernetes minion。工作负载也可以通过提供新 minion 的 IP 和正确的服务端口访问。试试吧！

15.4 用 Docker 实现平台即服务

如果你已经进入 IT 行业的平台即服务 (PAAS) 领域，那么我们在本章中讨论的概念你可能已经很熟悉。Pivotal 的 Cloud Foundry 和其他 PaaS 解决方案运营原则与 Kubernetes 相同：

在任意群集节点上以容器方式部署软件，并代替管理员管理可用性。在对 Kubernetes 有了初步认识之后，可以试验基于 Docker 的 PaaS 平台，如 ActiveState Stackato、Deis 和 Flynn。如果只对其他容器管理解决方案感兴趣，Apache Mesos (Twitter 和其他公司使用) 是另一个流行的解决方案。

15.5　小结

使用 Kubernetes 等解决方案管理时，群集上的应用容器部署得到了简化。虽然 Kubernetes 包含多个组件，但使用具备 Cloud-Config 的 CoreOS 时，基础设施的设置可以自动化。随着 VMware 和 Docker 及 Google 在容器管理上的合作日趋紧密，可以预期这一领域会有令人激动的发展。

参考文献

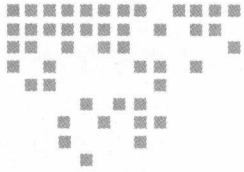
[1] Kubernetes GitHub repository and documentation: https://github.com/GoogleCloudPlatform/kubernetes

[2] The Raft consensus algorithm: http://raftconsensus.github.io/

第七部分 *Part 7*

DevOps 工具链

- 第 16 章　使用 Razor 配给服务器
- 第 17 章　ELK——Elasticsearch、Logstash 和 Kibana 简介
- 第 18 章　用 Jenkins 实现持续集成



Chapter 16 第 16 章

使用 Razor 配给服务器

Razor 是一种结合了服务器安装、管理和配置工具的自动化配给工具。它可以帮助你从裸机或者没有安装任何操作系统的虚拟机（VM）开始，最终得到由 Puppet 或者 Chef 管理的全配置节点。

本章包含如下主题：

- Razor 工作原理
- Razor 的使用
- 使用 Razor API
- Razor 组件
- Razor 安装

16.1 Razor 的工作原理

Razor 结合了 iPXE、TFTP、DHCP 和 DNS 的能力，提供全自动化部署解决方案。除了极其强大之外，它还提供 API 以自动化所有任务。尽管自身已经很有用，Razor 还通过 Chef 和 Puppet 代理提供与配置工具的集成。这些工具全部整合起来形成了库存和服务器生命期管理的一种机制。

Razor 在自己的服务器上以 Web 应用（TorqueBox）的形式运行，所有支持工具配置为在同一台服务器或者独立服务器上。Razor 应该拥有自己的网络，且新机器连接到它。当新服务器（配置为网络引导）连接到 Razor 网络时，发生如下的步骤：

- 发现：Razor 检测到新节点，然后以 Razor 微内核引导节点，并收集有关服务器的信息（事实），检查节点的特性。
- 注册：如果服务器的事实匹配现有的某个标签，标签与 Razor 服务器上的策略比较。第一个匹配的策略应用到节点。
- 安装：根据应用的策略，用某种操作系统安装服务器并进行配置。
- 移交：如果策略包含代理信息，服务器被移交给 Chef 或者 Puppet。

图 16-1 帮助你了解通过 Razor 的实际安装过程，从图中可以看到网络引导的节点已经连接到 Razor，正在加载微内核。

提示　可以使用 Vagrant 在虚拟环境中安装 Razor，以试验本章中的例子。这个单 VM Vagrant 环境将提供配置了 dnsmasq 的 Razor：https://github.com/eglute/razor-server-vagrant。

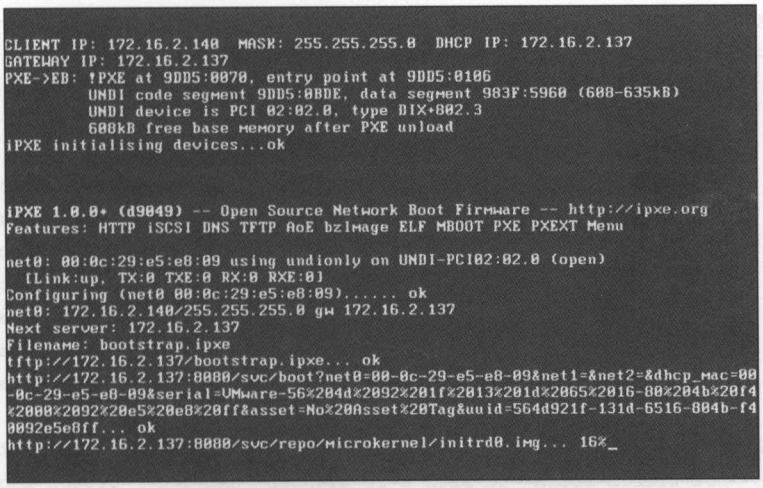


图 16-1　服务器正在下载微内核

在获取微内核之后，节点将注册到 Razor 并引导。图 16-2 展示了注册的节点、关联的任务和安装文件的加载。

最后，一切就绪之后，服务器将安装 OS。在图 16-3 中，节点用 Ubuntu 中定义的 Ubuntu preseed 文件安装。

注意　Razor 的当前版本是第二代 Razor，以 JRuby 编写，运行于 TorqueBox 上，使用 PostgreSQL 后端。Razor 的第一个版本基于 Ruby 和 Node.js，由 MongoDB 数据库支持，提供的功能与现行版本稍有不同。如果你是早期采用者，希望转移到当前版本，必须手工进行。从原始版本到下一版本没有任何升级 / 迁移路径。

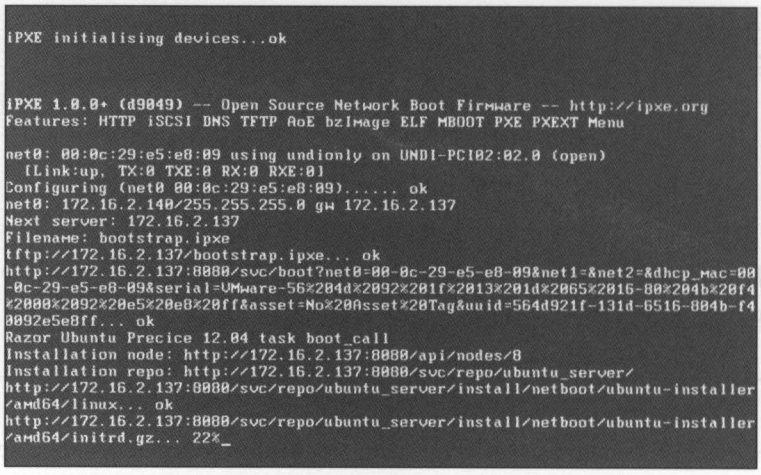


图 16-2 注册节点准备安装 Ubuntu 服务器

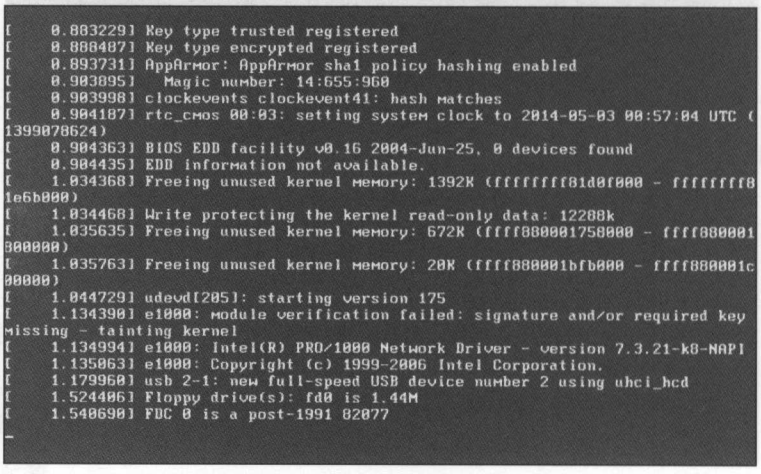


图 16-3 安装过程中的服务器

16.2 使用 Razor

如果已经安装和配置了 Razor，使用相当简单。Razor 遵循一组惯例，所有命令都易于使用。

注意 Razor 目前正在非常积极地开发中。访问命令参考指南（http://docs.puppetlabs.com/pe/3.2/razor_reference.html）可以了解最新的命令语法，因为从本书发行以来，这些语法可能稍有变化。

现在，我们假设你安装了 Razor 并且正常工作，在本章的后面将进一步介绍安装和所有组件。一旦安装了所有组件，就需要做一些基本的设置。首先，必须将 Razor 客户端指向 Razor 服务器：

```
$ razor --url http://$IP_ADDRESS:8080/api nodes
```

其中。$IP_ADDRESS 是 Razor 服务器的主机名或者 IP 地址：

```
$ razor --url http://razor:8080/api nodes
```

注意，Razor 客户端指向 Razor 服务器只需要进行一次，设置地址之后，键入 razor 或 razor -h 命令列出可用命令。程序清单 16-1 中展示了一个例子。

程序清单 16-1　Razor 客户端帮助和可用命令

```
$ razor
Usage: razor [FLAGS] NAVIGATION
    -d, --dump                  Dumps API output to the screen
    -u, --url URL               The full Razor API URL, can also be  set
                                with the RAZOR_API environment variable
                                (default http://localhost:8080/api)
    -h, --help                  Show this screen

    Collections:
        brokers
        nodes
        policies
        repos
        tags
        tasks

      Navigate to entries of a collection using COLLECTION NAME, for example,
      'nodes node15'  for the  details of a node or 'nodes node15 log' to see
      the log for node15

    Commands:
        add-policy-tag
        create-broker
        create-policy
        create-repo
        create-tag
        create-task
        delete-broker
        delete-node
        delete-policy
        delete-repo
        delete-tag
```

```
    disable-policy
    enable-policy
    modify-node-metadata
    modify-policy-max-count
    move-policy
    reboot-node
    reinstall-node
    remove-node-metadata
    remove-policy-tag
    set-node-desired-power-state
    set-node-ipmi-credentials
    update-node-metadata
    update-tag-rule
    Pass arguments to commands either directly by name ('--name=NAME')
    or save the JSON body for the  command  in a file and pass it with
    '--json FILE'.  Using --json is the only way to pass  arguments in
    nested structures such as the configuration for a broker.
```

如你所见，Razor 命令分为两个部分：集合和命令。集合和对象（事物）绑定，是 Razor 的一部分，命令是在集合中单独项目上执行的所有操作。输入如下格式的命令查看所有集合的列表：

```
$ razor <集合>
```

附加特定项目，可以查看每个集合的细节：

```
$ razor <集合> <项目名称>
```

所以，查看某个任务细节的命令如下：

```
$ razor tasks microkernel
```

操作在各种集合项目或者组合上执行。

16.2.1 Razor 集合和操作

Razor 中的集合是对象（绑定或者将要绑定到物理设备的虚拟资产）。有 6 种可能的集合；每一种集合中的单独项目上可以执行不同操作：

- 存储库
- 任务
- 代理
- 标签
- 策略
- 节点

任务是唯一预建的集合；其他集合必须在节点安装之前自行构建。

任务

Razor 任务描述在机器配给时应该执行的处理。每个任务有一个对应的 YAML 元数据文件和一组模板。Razor 自带多个内建任务，用户可以添加其他自定义任务。如程序清单 16-2 所示，输入 razor tasks 命令可以查看所有可用任务。

程序清单 16-2　Razor 内建任务

```
$ razor tasks
From http://localhost:8080/api/collections/tasks:
    id: "http://localhost:8080/api/collections/tasks/centos"
  name: "centos"
  spec: "/razor/v1/collections/tasks/member"
    id: "http://localhost:8080/api/collections/tasks/debian"
  name: "debian"
  spec: "/razor/v1/collections/tasks/member"
    id: "http://localhost:8080/api/collections/tasks/microkernel"
  name: "microkernel"
  spec: "/razor/v1/collections/tasks/member"
    id: "http://localhost:8080/api/collections/tasks/noop"
  name: "noop"
  spec: "/razor/v1/collections/tasks/member"
    id: "http://localhost:8080/api/collections/tasks/redhat"
  name: "redhat"
  spec: "/razor/v1/collections/tasks/member"
    id: "http://localhost:8080/api/collections/tasks/ubuntu"
  name: "ubuntu"
  spec: "/razor/v1/collections/tasks/member"
    id: "http://localhost:8080/api/collections/tasks/ubuntu_precise_amd64"
  name: "ubuntu_precise_amd64"
  spec: "/razor/v1/collections/tasks/member"
    id: "http://localhost:8080/api/collections/tasks/ubuntu_precise_i386"
  name: "ubuntu_precise_i386"
  spec: "/razor/v1/collections/tasks/member"
    id: "http://localhost:8080/api/collections/tasks/vmware_esxi"
  name: "vmware_esxi"
  spec: "/razor/v1/collections/tasks/member"
    id: "http://localhost:8080/api/collections/tasks/windows"
  name: "windows"
  spec: "/razor/v1/collections/tasks/member"
```

如果 Razor 中添加了对应的存储库，每个任务代表可安装的一个操作系统。任务和存储库在策略中被绑定在一起，为了查看任务的细节，你必须转向 Razor 安装目录下的 tasks 文

件夹。根据你的系统，该位置可能是 /opt/razor/tasks。程序清单 16-3 展示了安装 ESXi 示例的 YAML 元数据文件。

程序清单 16-3 用于 vmware_esxi 任务的 vmware_esxi.yaml 文件

```
---
os_version: 5.5
label: VMWare ESXi 5.5
description: VMWare ESXi 5.5
boot_sequence:
  1: boot_install
  default: boot_local
```

vmware_esxi 任务将利用 razor/tasks/vmware_esxi 目录下找到的 boot_install.erb 文件安装 ESXi 5.5。该文件包含自动化 iPXE 命令的脚本，可以在程序清单 16-4 中引用。

程序清单 16-4 ESXi 5.5 所用的 boot_install.erb 文件

```
#!ipxe
echo Razor <%= task.label %> task boot_call
echo Installation node: <%= node_url  %>
echo Installation repo: <%= repo_url %>

echo Booting ESXi installer via pxelinux to support COMBOOT modules fully
echo Unfortunately, this double bootloader is necessary to run the ESXi
installer

sleep 3
# These are the magic that direct pxelinux.0 to our HTTP based config...
set 209:string pxelinux_esxi.cfg
set 210:string <%= file_url('') %>
imgexec <%= file_url('pxelinux.0', true) %>

:error
prompt --key s --timeout 60000 ERROR, hit 's' for the iPXE shell, reboot in
60 seconds && shell || reboot
```

如果你想添加自己的自定义任务，必须复制现有任务的结构。关于编写自定义任务的更多细节，参考文档：http://docs.puppetlabs.com/pe/3.2/razor_tasks.html。

节点

节点是 Razor 中最重要的集合，因为它代表着 Razor 管理的机器。节点包含所有来自机器的“事实”。输入 razor nodes 命令可以查看当前注册到 Razor 服务器的节点（参见程序清单 16-5）。

程序清单 16-5　当前注册到 Razor 的节点列表

```
$ razor nodes
From http://localhost:8080/api/collections/nodes:

    id: "http://localhost:8080/api/collections/nodes/node1"
  name: "node1"
  spec: "/razor/v1/collections/nodes/member"

    id: "http://localhost:8080/api/collections/nodes/node2"
  name: "node2"
  spec: "/razor/v1/collections/nodes/member"
```

检查单个节点的细节可提供许多信息。这是因为 Razor 收集和存储了关于每个服务器的许多标识信息。这些事实用于识别节点。程序清单 16-6 列出了单独节点的可用细节。

程序清单 16-6　单独节点细节

```
$ razor nodes node1
From http://localhost:8080/api/collections/nodes/node1:

            id: "http://localhost:8080/api/collections/nodes/node1"
          name: "node1"
          spec: "/razor/v1/collections/nodes/member"
       hw_info:
                 asset: "no asset tag"
                   mac: ["00-0c-29-cd-fc-ee"]
                serial: "vmware-56 4d 61 fd b3 73 ee 95-8a d3 dd 4c 07 cd fc
ee"
                  uuid: "564d61fd-b373-ee95-8ad3-dd4c07cdfcee"
    dhcp_mac: "00-0c-29-cd-fc-ee"
      policy:
                  id: "http://localhost:8080/api/collections/policies/
ubuntu_one"
                name: "ubuntu_one"
                spec: "/razor/v1/collections/policies/member"
         log:
              log => http://localhost:8080/api/collections/nodes/node1/log
        tags: [
                    id: "http://localhost:8080/api/collections/tags/
ubuntu_small"
                  name: "ubuntu_small"
                  spec: "/razor/v1/collections/tags/member"
              ]
       facts:
                          architecture: "x86_64"
```

```
                    blockdevice_sda_size: 21474836480
                  blockdevice_sda_vendor: "VMware,"
  interfaces: "eno16777736,eno33554960,lo"
                   ipaddress_eno16777736: "172.16.2.141"
                  macaddress_eno16777736: "00:0c:29:cd:fc:ee"
                    netmask_eno16777736: "255.255.255.0"
< … >
    processorcount: "1"
                                uniqueid: "007f0100"
                                 virtual: "vmware"
                              is_virtual: "true"
     metadata:
                  ip: "172.16.2.141"
       state:
                   installed: "ubuntu_one"
                  installed_at: "2014-04-23T11:50:31-07:00"
                       stage: "boot_local"
     hostname: "host"
  root_password: "secret"
   last_checkin: "2014-04-23T11:31:06-07:00"
```

节点细节（参见程序清单 16-6）包含服务器的物理特性（事实）、节点创建方式的相关信息（应用的策略）、最后一次登记的时间以及查找日志信息的方法。

警告 因为 Razor 根据物理特性识别节点，更新在 Razor 中注册的节点硬件可能使 Razor 产生混乱。删除或者更换主 NIC 可能使 Razor 认为是一个新的节点。

除了默认事实之外，Razor 允许用户添加每个节点的自定义元数据。这些自定义元数据使用户可以添加业务或者运营特定的标识数据。添加自定义元数据的方法参见命令参考指南：http://docs.puppetlabs.com/pe/3.2/razor_reference.html#modify-node-metadata。

节点的日志信息将包含节点创建中执行的不同步骤（参见程序清单 16-7）。

程序清单 16-7 查看节点活动日志

```
$ razor nodes node1 log
From http://localhost:8080/api/collections/nodes/node1/log:

  timestamp: "2014-04-23T11:27:18-07:00"
      event: "boot"
       task: "microkernel"
   template: "boot"
       repo: "microkernel"
   severity: "info"
```

```
< … >
timestamp: "2014-04-23T11:50:30-07:00"
    event: "store_metadata"
     vars:
            update:
                    ip: "172.16.2.141"
  severity: "info"

 timestamp: "2014-04-23T11:50:31-07:00"
    event: "stage_done"
    stage: "finished"
  severity: "info"
```

安装节点有多个步骤，日志将显示过程中的状态和失败（如果有的话）。一旦节点安装结束，节点的 /tmp 目录中也会创建一个日志。razor_complete.log 可能包含安装中所出现错误的有用信息。

注意　目前，在 Razor 中添加节点的方法是，安装新服务器并通过微内核注册。可以使用 noop 任务注册现有服务器，但是未来会有明确的方法在 Razor 中添加现有服务器，最新的功能请查看发行说明。

任何节点都可以从 Razor 中删除，只需要执行删除命令 razor delete-node --name < 节点名称 >，就可以删除节点，参见程序清单 16-8 中的演示。

程序清单 16-8　删除现有节点

```
$ razor delete-node --name node3
From http://localhost:8080/api:

  result: "node destroyed"
```

如果同一个服务器再次引导，节点将在 Razor 中重新创建。

要重新安装现有节点，可以执行重启命令 razor reinstall-name --name <name4>，程序清单 16-9 中有一个演示。

程序清单 16-9　重新安装现有节点

```
razor reinstall-node --name node4
From http://localhost:8080/api:

  result: "node unbound from ubuntu_two and installed flag cleared"
```

节点重启之后，将再次加载微内核并重新安装。如果有适用的新规则，将在安装期间应用。

IPMI 支持

目前，Razor 支持一组 IPMI 功能。如果硬件支持，可以通过 Razor 重启并设置节点状态。首先，必须设置 IPMI 凭证，创建包含相关细节的 ipmi.json 文件，然后用它设置节点上的凭据，如程序清单 16-10 所示。

程序清单 16-10 设置 IPMI 凭据的 ipmi.json 文件

```
{
  "name": "node1",
  "ipmi-hostname": "node1",
  "ipmi-username": "root",
  "ipmi-password": "secret"
}
```

节点上 IPMI 数据的设置通过执行 razor set-node-ipmi- credentials --json <json 文件名 > 命令完成，如程序清单 16-11 所示。

程序清单 16-11 在节点上设置 IPMI 凭据

```
$ razor set-node-ipmi-credentials --json ipmi.json
From http://localhost:8080/api:

  result: "updated IPMI details"
```

设置 IPMI 凭据之后，可以通过 Razor 重新启动并设置节点的电源状态。

16.2.2 构建 Razor 集合

为了运行 Razor，必须添加一些初始信息。按照如下顺序构建对象：

- 添加存储库
- 添加代理
- 添加标签
- 添加策略

对象的创建可以通过在命令行上传递属性或者创建 JSON 文件并将该文件作为参数调用来完成。使用 JSON 文件比在命令行上列出所有信息容易得多，尤其是在创建策略时。

添加存储库

存储库是保存安装节点实际数据的容器，它是一个 ISO 文件或者指向 ISO 文件的链接。创建存储库（repo）的基本语法如下：

```
$ razor create-repo --name=<存储库名称> --iso-url <URL>
```

存储库名称可以任意选择。存储库可以通过传递 URL 链接或者使用本地文件创建，当

ISO 文件在远程机器上时，创建存储库可能需要花费一些时间，因为下载需要时间。检查日志文件，查看存储库创建是否结束。

```
$ razor create-repo --name=ubuntu_server --iso-url http://releases.
ubuntu.com/precise/ubuntu-12.04.4-server-amd64.iso
```

如果想要使用本地 ISO，可以使用相同的命令行语法指向它：

```
$ razor create-repo --name=ubuntu_server --iso-url file:///root/ubuntu-
12.04.4-server-amd64.iso
```

注意，文件和路径必须可用于运行 Razor 的用户。

添加存储库对于每个映像只需要进行一次。

创建存储库之后，ISO 文件将解压到 /var/lib/razor/repo-store 或者类似的目录下。在 Puppet 企业版中，默认路径为 /opt/puppet/var/razor/repo/。

添加代理

Razor 代理是将配给的服务器移交给配置管理工具的手段。代理由配置文件和脚本组成。脚本在所有主要安装完成之后运行，并在机器上安装 Puppet 或者 Chef 客户端。目前有 4 种代理类型：

- Noop（noop）：如果不想通过 Razor 在新安装的服务器上进行更多自动化，则使用 Noop 代理。
- Chef（chef）：Chef 代理在新安装的服务器上安装 Chef 客户端，注册到 Chef 服务器，并执行初始运行列表。
- Puppet（puppet）：Puppet 代理安装 Puppet 代理并注册到 Puppet 主服务器。
- Puppet 企业版（puppet-pe）：Puppet 企业版代理配置新节点，注册到 Puppet 主服务器。

添加简单的 Noop 代理只需要使用 create-broker 命令：

```
$ razor create-broker --name=noop --broker-type=noop
```

添加代理的语法如下：

```
$ razor create-broker --name <名称> --broker-type <类型> --configuration
<配置>
```

其中配置必须使用 JSON 格式。

除非添加 Noop 代理，否则使用 JSON 格式的配置文件要简单得多：

```
$ razor create-broker --json <file.json>
```

每个代理配置文件由 3 部分组成：名称、代理类型和配置。配置特定于代理类型，不同代理的配置不同。

程序清单 16-12 展示了一个 Puppet 代理 JSON 配置文件。

程序清单 16-12 创建 Puppet 代理的 puppetbroker.json 文件示例

```
{
  "name": "puppet",
  "configuration": {
     "server": "puppet.example.org",
     "environment": "production"
  },
  "broker-type": "puppet"
}
```

要创建 Puppet 代理，所需的就是传递一个 JSON 配置文件。可以按照需要命名 JSON 文件，在程序清单 16-13 的例子中，我们称其为 puppetbroker.json。

程序清单 16-13 创建 Razor 代理

```
$ razor create-broker --json puppetbroker.json
From http://localhost:8080/api:

    id: "http://localhost:8080/api/collections/brokers/puppet"
  name: "puppet"
  spec: "/razor/v1/collections/brokers/member"
```

创建 Chef 代理更加复杂，因为需要配置选项。在 Razor 安装目录下的 broker 目录中提供了一个配置文件样板 sample_chef_broker.json，如程序清单 16-14 所示。

程序清单 16-14 创建 Razor Chef 代理的 sample_chef_broker.json 文件

```
{
   "name": "openstack_chef_broker",
   "configuration": {
        "install_sh": "http://opscode.com/chef/install.sh",
        "version_string": "11.4.4",
        "chef_server_url": "https://chef.example.com:443",
        "validation_client_name": "chef-validator",
        "run_list": "role[allinone]",
        "environment": "openstack",
        "chef_client": "chef-client",
        "validation_key": "MIIok0h+mSHr1Pbi+v2950H1/7mzd71hXAcRHbmrdy-
tAR+RjgtyibkkZTdSxjLP6o2qEuji0I7DL8cckHG56AvuAKhYoadu+9J/2ahmYr18CSqOmg4S-
bh6CPm5edpqaAVTbE3Ec+1wN0IMl8KWtmGCrjpXzH1MDdaLZpIIYqC9DUVgLbd/
i61hbiiYlky5wPdIKlKwilRl7alfsGKTUrVq1DuYiIZsDYrnY-LOERbzU6yUxWSIasfVP-
JGpT9LvstChFjyjv/73etmhXwwIDAQABAoIBAHH1p1upll-VJNMSAhSmLzfTe6Q9nT8unRF-
H1egcsni8dPXYyVzDQ1ztV3RFNDLjP01ZThqkjoA4TnkHvRnC90yR049eQNxn+7pctcTNW61aA-
glomMhBcFt7LlcDiiXfD3dVhoIDv4ijpfsNCDvEVain9Dv+krjGVP+dWKLKr2azZbrAnyJKJjKb-
8DkwFET7UADFgUiCwJCW7RyAAxL40WE1hyYGq03cum5+M+2MGTcCiiAumqIRjYyTioNl73zf/
```

```
ckBKRBNb6DkVGa1drAwXqMtA0MAaJlQBoJJF9SehSf7rAKVMvQ+esS9kNXZhECgYEA28FK-
bU3K4mU2khZ8lJgomfelWV48sjeIH3OEuLK7WdKg2LAnNajiLAOHYi0egKiMJPnNyZ8u-
vOY0JABLHQXv10QKaNlHGqy+aRiRSb1Ya6SlbCg8hMoT29yWGa8k7AclLJSn20/dg0GAEQtegi/
k3EEZq0CgYEA148QtVh4ng80d9P81rJEDkraAkKUNJPHBSKFqo1rNwSDsyxxTEXIzZSPYS1m-
jsdB4IxraiNm3WDhb+EldcSlXvmBDjYJck1WyLCw1JuyAS2VCEtdc38aPWGYZ157de-
cuzkR8CU2TKw48KGSRTtwL8yYqTknqpAmnqf/KkWfVNi8CgYEA1a0AnX+Cwtf/
U9UhlarN78fosxj6k5+DcHF2n9DKcvBnDctalMZ+BKX5wfB1NEyu2FU9T8rEO1DragYPA01+h-
CXYqVJ73OgSzUXiH31IuIID9u/0LyseTWOWoIxWJzdTh44xJ+wJlwNuYXxgjgVGaVZk-
2niXWTLxtTNEdCz0RpECgYAlholhGIq+8WSv656bfaMtXciAFjkYwhUmhrEAVOgyRr3qpjT/
EzL23wLq5u1ws617OtbEPm984I2+XVKZIuerFgJqh+uzE1WlUGUoTgZ6AAZu0DfvkFPwFZpjfGY/
y0QxsmhpcjDJkQvV+FP3h4UpCh7ZTDLl5axjgt0v3QSYDwKBgFde/5TO8N8U6lHr1YX+yY8w-
bQ9sVWPU8qruL5Qx11an1tm9Ja1Wbg8Sn0/A7h7Y331V4cDmVraUreULiTQwSO7N26IxQ3Rg/
MQG3szUgP0MWYmjuG0c8zaFB73rnBpZ8xakF/xcRTt2Pb62dkw1VqFhzNc50bN+QvGmtE-
osIB9z"

},
    "broker-type": "chef"
}
```

从创建代理所需的 JSON 文件（程序清单 16-14）中可以看到，Chef 代理配置选项要复杂得多。我们将使用代理目录中找到的代理需求配置文件（如程序清单 16-15 所示），解释不同的选项。这个文件描述配置需求。

程序清单 16-15　Chef 代理配置文件 configuration.yaml

```
---
install_sh:
  description: "Omnibus installer script URL. (example: http://mirror.
example.com/install.sh) default: http://opscode.com/chef/install.sh"
version_string:
  description: "Chef version (used in gem install). (example: 11.4.4)"
validation_key:
  description: "Contents of the validation.pem file. Since this is json
file, please make it one line no spaces..."
chef_server_url:
  description: "URL for the Chef Server. (example: https://mychefserver.
com:443)"
validation_client_name:
  description: "Validation client name. (example: myorg-validator) default:
chef-validator"
run_list:
  description: "Optional run_list of common base roles. (example:
role[base],role[another])"
environment:
  description: "Chef environment in which the chef-client will run. (exam-
ple: production). Use _default if not using any specific environments."
```

```
chef_client:
  description: "An alternate path to the chef-client binary. (example: /
usr/local/bin/chef-client) default: chef-client"
```

不同的选项保证 Chef 代理可以完全自定义，以适应环境需求。

注意 需要注意的是，Razor 代理的 validation_key 值总是在同一行中，这是因为配置使用 JSON 格式的要求。代理的所有配置选项集中起来保存在 Razor 数据库的一行中。通常，校验密钥应该包含换行符。Chef 代理取得 validation_key 参数并重新插入换行符，从而在客户端重建正常的 validation.pem 文件结构。

只需为用于 Razor 的每个配置环境创建一个代理。输入命令 razor brokers 可以查看创建的所有代理，如程序清单 16-16 所示。

程序清单 16-16 Chef 代理配置文件 configuration.yaml

```
$ razor brokers
From http://localhost:8080/api/collections/brokers:

  id: "http://localhost:8080/api/collections/brokers/noop"
name: "noop"
spec: "/razor/v1/collections/brokers/member"

  id: "http://localhost:8080/api/collections/brokers/puppet"
name: "puppet"
spec: "/razor/v1/collections/brokers/member"
```

每个代理将会保留使用它们的策略数量。输入 razor brokers < 代理名称 > 命令查看特定代理的详细情况，如程序清单 16-17 所示。

程序清单 16-17 查看 Puppet 代理的详细情况

```
$ razor brokers puppet
From http://localhost:8080/api/collections/brokers/puppet:

            id: "http://localhost:8080/api/collections/brokers/puppet"
          name: "puppet"
          spec: "/spec/object/broker"
 configuration:
                      server: "puppet.example.org"
                 environment: "production"
   broker-type: "puppet"
      policies:
```

```
                id: "http://localhost:8080/api/collections/brokers/
puppet/policies"
              name: "policies"
             count: 0
```

在本书编著时，没有任何命令能够列出所有可用代理。但是，如果你有 Razor 服务器的访问权限，可以访问 Razor 安装目录下的代理目录，查看哪些代理可用。类似地，每个代理都有自己的 configuration.yaml 文件，有助于确定每个代理接受的参数。

如果想使用不存在代理的不同配置工具，可以按照 Razor GitHub 网站上的文档添加自定义代理：https://github.com/puppetlabs/razor-server/wiki/Writing-broker-types。

添加策略

策略以合乎逻辑的安排组合存储库、任务、标签和代理，确定新节点上所要安装的系统和安装的次数。策略可以启用和禁用，而且会记录自身应用到某个节点的次数。除了编排所有其他信息之外，策略还为节点设置主机名和密码。目前，密码以普通文本保存，所以建议通过配置文件改变新安装节点的密码。

要创建策略，首先创建 policy.json 文件（参见程序清单 16-18），然后使用 razor create-policy 命令。

程序清单 16-18　policy.json

```
{
  "name": "ubuntu_one",
  "repo": { "name": "ubuntu_server" },
  "task": { "name": "ubuntu" },
  "broker": { "name": "noop" },
  "enabled": true,
  "hostname": "host${id}",
  "root_password": "secret",
  "max_count": "20",
  "tags": [{ "name": "ubuntu_small", "rule": ["=", ["num", ["fact",
"processorcount"]], 1]}]
}
```

程序清单 16-18 展示了如下策略属性：

- name: 新创建策略的名称，必须唯一。
- repo：用于安装节点的存储库名称，必须已经创建。
- task：将要使用的任务名称。注意，任务和存储库必须是同一类型；也就是说，centos 安装要使用 centos 任务，依此类推。
- broker：创建的代理名称，必须已经存在。
- enabled：策略是否应该启用。如果设置为 false，Razor 将在搜索安装节点的匹配策略

时忽略这一策略。

- hostname：用于在新安装节点上设置主机名。${id} 表达式将从数据库中求取一个数字，每个节点将递增 1。在这个例子中，第一个节点的主机名为 node1，第二个为 node2，依此类推。
- root_password：在新节点上设置的根密码。在安装节点之后更改，因为该密码保存为普通文本。
- max_count：使用这个策略安装的服务器数量。如果只想用特殊策略安装一台服务器，将其设置为 1。
- tags：确定本策略适用于哪类节点。以后也可以为策略添加更多标签。

如程序清单 16-19 所示，执行 razor create-policy --json < 文件名 > 命令可以添加一个策略。

程序清单 16-19 创建策略

```
$ razor create-policy --json policy.json
From http://localhost:8080/api:

    id: "http://localhost:8080/api/collections/policies/ubuntu_one"
  name: "ubuntu_two"
  spec: "/razor/v1/collections/policies/member"
```

如程序清单 16-20 所示，执行 razor policies 命令列出所有可用策略。

程序清单 16-20 列出现有策略

```
$ razor policies
From http://localhost:8080/api/collections/policies:

    id: "http://localhost:8080/api/collections/policies/ubuntu_one"
  name: "ubuntu_one"
  spec: "/razor/v1/collections/policies/member"

    id: "http://localhost:8080/api/collections/policies/ubuntu_two"
  name: "ubuntu_two"
  spec: "/razor/v1/collections/policies/member"
```

查看策略可以检查其细节，如程序清单 16-21 所示，使用 razor policies < 策略名称 > 命令查看策略。

程序清单 16-21 查看策略细节

```
$ razor policies ubuntu_one
From http://localhost:8080/api/collections/policies/ubuntu_one:

          id: "http://localhost:8080/api/collections/policies/ubuntu_
```

```
one"
            name: "ubuntu_one"
            spec: "/razor/v1/collections/policies/member"
            repo:
                    id: "http://localhost:8080/api/collections/repos/
ubuntu_server"
                  name: "ubuntu_server"
                  spec: "/razor/v1/collections/repos/member"
            task:
                    id: "http://localhost:8080/api/collections/tasks/
ubuntu"
                  name: "ubuntu"
                  spec: "/razor/v1/collections/tasks/member"
          broker:
                    id: "http://localhost:8080/api/collections/brokers/
noop"
                  name: "noop"
                  spec: "/razor/v1/collections/brokers/member"
        enabled: true
      max_count: 20
  configuration:
                  hostname_pattern: "host"
                  root_password: "secret"
            tags: [
                    id: "http://localhost:8080/api/collections/tags/
ubuntu_small"
                  name: "ubuntu_small"
                  spec: "/razor/v1/collections/tags/member"
]
           nodes:
                    id: "http://localhost:8080/api/collections/policies/
ubuntu_one/nodes"
                  name: "nodes"
                 count: 2
```

注意，细节不仅包含创建策略时添加的信息，还包括节点计数。

添加标签

标签由名称和规则组成。它们在策略中使用，通过评估服务器的特性决定是否应该将策略应用到节点。特性应该是从尺寸到以太网卡类型或 MAC 地址的任何信息。

标签的规则由一个运算符和一个或者多个参数组成。应用到策略时，标签将以 if/then 表达式的方式求值：如果服务器的细节与该标签相符，则应用策略。

标签可以单独创建，也可以作为策略的一部分。为了创建用于唯一 MAC 地址的策略，我们首先创建一个 JSON 文件 macs_tag.json，如程序清单 16-22 所示。

程序清单 16-22 macs_tag.json 文件

```
{
    "name": "unique_macs",
    "rule": [
        "in",
        [
            "fact",
            "macaddress"
        ],
        "de:ea:db:ee:f0:00",
        "de:ea:db:ee:f0:01"
    ]
}
```

然后，使用 razor create-tag --json < 文件名 > 命令就可以简单地创建标签，如程序清单 16-23 所示。

程序清单 16-23 创建标签

```
$ razor create-tag --json macs_tag.json
From http://localhost:8080/api:

    id: "http://localhost:8080/api/collections/tags/unique_macs"
  name: "unique_macs"
  spec: "/razor/v1/collections/tags/member"
```

创建 JSON 格式文档时，有一些很方便的校验和格式化工具，既有命令行工具，也有 Web 应用。http://jsonlint.com/ 就是这样一种工具。

标签也可以添加到策略或从中删除。为在策略中添加现有的标签，我们将创建另一个 JSON 文件 another_tag.json，如程序清单 16-24 所示。

程序清单 16-24 another_tag.json 文件

```
{
  "name": "ubuntu_one",
  "tag" : "unique_macs"
}
```

这里，“name”是现有策略的名称，“tag”是现有标签的名称。类似地，我们将创建一个文件，新标签同时在其中创建和添加。比较程序清单 16-24 和 16-25。

程序清单 16-25　another_tag.json 文件的另一种版本

```
{
    "name": "ubuntu_one",
    "tag": "processor_count_2",
    "rule": [
        "=",
        [
            "fact",
            "processorcount"
        ],
        "2"
    ]
}
```

在现有策略中添加新标签时，任何一种格式都有效：

```
$ razor add-policy-tag --json another_tag.json
```

现在，如果我们检查策略的标签部分，将会看到该策略有两个新标签，如程序清单 16-26 所示。

程序清单 16-26　更新后的 ubuntu_one 策略的标签部分：

```
tags: [
                    id: "http://localhost:8080/api/collections/tags/
ubuntu_small"
                  name: "ubuntu_small"
                  spec: "/razor/v1/collections/tags/member"

                    id: "http://localhost:8080/api/collections/tags/
unique_macs"
                  name: "unique_macs"
                  spec: "/razor/v1/collections/tags/member"

                    id: "http://localhost:8080/api/collections/tags/
processor_count_2"
                  name: "processor_count_2"
                  spec: "/razor/v1/collections/tags/member"
              ]
```

标签可以类似方式更新和删除。但是，做这些工作时必须格外小心，因为会影响现有策略。

16.3　使用 Razor API

目前，所有示例都是用 Razor 客户端完成的。但是，客户端只是一个“瘦”包装器，向

服务器发出 REST 风格调用。你可能已经观察到，这些命令的输出和每个集合项目的细节包含带有 URL 的 ID。这不是巧合，实际上，在执行 Razor 命令时，可以向任何命令传递 -d 标志，就会得到原始 JSON 格式的服务器返回输出。而且，没有必要使用客户端：API 的结构使得编写自己的自动化工具与 Razor 服务器交互很容易。例如，如果通过客户端调用列出所有标签，输出如程序清单 16-27 所示。

程序清单 16-27 使用 Razor 客户端得到的 ubuntu_small 标签细节

```
$ razor tags ubuntu_small
From http://localhost:8080/api/collections/tags/ubuntu_small:
        id: "http://localhost:8080/api/collections/tags/ubuntu_small"
      name: "ubuntu_small"
      spec: "/razor/v1/collections/tags/member"
      rule: [
              "="
              ["num", ["fact", "processorcount"]]
              1
            ]
     nodes:
             id: "http://localhost:8080/api/collections/tags/ubuntu_
small/nodes"
           name: "nodes"
          count: 3
  policies:
             id: "http://localhost:8080/api/collections/tags/ubuntu_
small/policies"
           name: "policies"
          count: 1
```

突出显示的 URL 可以直接用来调用 Razor 服务器。程序清单 16-28 中的输出通过 API 发出完全相同的调用，查看 ubuntu_small 标签的细节。

程序清单 16-28 查看 ubuntu_small 标签细节的 API 调用

```
$ curl http://localhost:8080/api/collections/tags/ubuntu_small
{"spec":"http://api.puppetlabs.com/razor/v1/collections/tags/
member","id":"http://localhost:8080/api/collections/tags/ubuntu_
small","name":"ubuntu_small","rule":["=",["num",["fact","processorcount"]],
1],"nodes":{"id":"http://localhost:8080/api/collections/tags/ubuntu_small/
nodes","count":3,"name":"nodes"},"policies":{"id":"http://localhost:8080/
api/collections/tags/ubuntu_small/policies","count":1,"name":"policies"}}
```

类似地，可以直接从自己的工具中使用 API 调用 Razor 服务器。因为大部分 Razor 客户端命令要求 JSON 输入，API 调用同样用相同的 JSON 输入创建和修改集合项目。

16.4　Razor 组件

Razor 由多个组件和应用程序工作必需的支持工具构成。

16.4.1　Razor 服务器

Razor 服务器是运营的大脑，是 Razor 的主组件，它以 JRuby 编写，运行于 TorqueBox 应用程序平台。TorqueBox 基于 JBoss 应用服务器，提供内建群集、高可用性和负载平衡。Razor 服务器需要 PostgreSQL 作为后端，在一台服务器上支持开发 / 测试 / 生产模式。用户通过 Razor 客户端或者直接通过 RESTful API 和服务器交互。

16.4.2　Razor 微内核

Razor 微内核是在节点上引导并将其记入库存清单的小型 Linux 映像。微内核的当前版本基于 Fedora，使用 MAC 地址作为唯一标识符。发现之后，如果服务器没有配置匹配的策略，将为微内核提供登录屏幕。但是，通常不需要登录到微内核本身。微内核是 64 位的映像，所以只支持 64 位机器。

Razor 只能配给 64 位机器。

16.4.3　Razor 客户端

Razor 客户端是命令行客户端，可以在任何从网络上访问 Razor 服务器的机器上安装。客户端必须单独安装，为了安全，最好将客户端安装在 Razor 服务器的同一台机器上，因为客户端本身目前没有提供任何身份验证。

Razor 客户端应该使用与 Ruby 1.9.3 兼容的 Ruby 解释程序。虽然 Razor 客户端在 JRuby 下可以运行得很好，但是将会非常缓慢。使用 Ruby 环境管理器（如 rbenv 或者 RVM）在不同版本之间切换。

Razor 客户端可以 Ruby gem 的形式安装。如果在同一台机器上安装 Razor 服务器和客户端，要注意别对客户端使用 JRuby。如果注意到客户端返回极慢，检查 Ruby 版本；JRuby 会使客户端变得很慢。

16.5　安装 Razor

Razor 可以多种方式安装。如果使用 Puppet 企业版，则自带 Razor。但是，必须在完全

独立的环境中测试 Razor，才能将其部署到生产环境。应该首先在虚拟环境中测试 Razor，了解其真实工作状况，对其能力有所感觉。

16.5.1 PE Razor

Razor 可以几种不同方式安装，如果使用 Puppet 企业版，Razor 已经可用；但是，仍然需要安装其他组件。默认没有安装的组件是 DHCP 服务器、DNS 服务器和映像存储库。

16.5.2 Puppet 安装

如果 Puppet 已经运行，安装 Razor 只要两行命令：

```
puppet module install puppetlabs/razor
puppet apply -e 'include razor'
```

但是，仍然需要自行安装和配置数据库。

16.5.3 从来源安装

如果没有使用 Puppet，就必须从 Razor 来源安装。Razor GitHub 网站提供安装指南：https://github.com/puppetlabs/razor-server/wiki/Installation。除非你很喜欢冒险，否则不要从主干安装，而采用最新的发行版本。从某个发行版本安装与从源代码安装相同。最新版本可以在这里找到：https://github.com/puppetlabs/razor-server/releases。

提示 如果打算设置一个测试环境，从 Vagrant 开始，有多种预建的 Vagrant 环境。以下 Vagrant 环境是一个运行 Puppet 企业版、DHCP 和 Razor VM 的 3 服务器环境：https://github.com/npwalker/pe-razor-vagrant-stack。这个单 VM 环境将提供配置了 dnsmasq 的 Razor 最新版本：https://github.com/eglute/razor-server-vagrant。

16.5.4 人工安装发行版本

如果没有使用 Puppet 或者 Puppet 企业版，那么这就是最简单的 Razor 安装方式。遵循 Razor wiki 上的安装指南：https://github.com/puppetlabs/razorserver/wiki/Installation。

16.5.5 其他服务

为使 Razor 正常工作，需要其他服务。Razor 的后端是 PostgreSQL，如果它没有作为 Razor 安装的一部分，可能需要安装和配置。

Razor 需要访问在安装中使用的 OS 映像。Razor 知道如何访问本地文件和远程文件。如果使用远程映像，Razor 服务器需要 HTTP 访问以读取它们。在读取映像之后，Razor 展开它并本地保存到 repo 目录。

Razor 还需要 DHCP 和 DNS 服务。IP 地址通过 DHCP 分配给新安装的节点。你可以使用自己选择的 DHCP 服务器，如果环境很小，dnsmasq 就是合适的 DHCP 和 DNS。

dnsmasq 配置文件（dnsmasq.conf）如程序清单 16-29 所示。

程序清单 16-29　样板 dnsmasq.conf 文件

```
server=$IP_ADDRESS@eth1
interface=eth1
no-dhcp-interface=eth0
domain=razor.local
# conf-dir=/etc/dnsmasq.d
# This works for dnsmasq 2.45
# iPXE sets option 175, mark it for network IPXEBOOT
dhcp-match=IPXEBOOT,175
dhcp-boot=net:IPXEBOOT,bootstrap.ipxe
dhcp-boot=undionly.kpxe
# TFTP setup
enable-tftp
tftp-root=/var/lib/tftpboot
dhcp-range=$IP_ADDRESS,$IP_RANGE,12h
dhcp-option=option:ntp-server,$IP_ADDRESS
```

注意，根据环境和安装，该文件可能不同。在本例中，$IP_ADDRESS 是 Razor 可以分配给新节点的起始地址，$IP_RANGE 是最后一个地址。

警告　将 Razor 部署到生产环境之前，先进行测试。

如果配置不当，Razor 可能影响现有基础设施，重启在 Razor 网络上的服务器可能造成现有机器重新安装。

故障检修

安装和配置所有组件之后，如果新服务器仍然无法启动，作如下检查：

- Razor 是否运行？因为 Razor 运行于 TorqueBox 上，检查是否运行 Java 进程（参见程序清单 16-30）。

程序清单 16-30　检查 Java 进程是否运行

```
ps -ef | grep java
root      2451  2420  0 Apr23 pts/0    00:04:15 java -D[Standalone] -server
-XX:+UseCompressedOops -Xms64m -Xmx512m -XX:MaxPermSize=256m -Djava.
net.preferIPv4Stack=true -Djboss.modules.system.pkgs=org.jboss.byteman
-Djava.awt.headless=true -Dorg.jboss.boot.log.file=/root/.rbenv/versions/
jruby-1.7.8/lib/ruby/gems/shared/gems/torquebox-server-3.0.1-java/jboss/
standalone/log/server.log -Dlogging.configuration=file:/root/.rbenv/
versions/jruby-1.7.8/lib/ruby/gems/shared/gems/torquebox-server-3.0.1-
```

```
java/jboss/standalone/configuration/logging.properties -jar /root/.rbenv/
versions/jruby-1.7.8/lib/ruby/gems/shared/gems/torquebox-server-3.0.1-java/
jboss/jboss-modules.jar -mp /root/.rbenv/versions/jruby-1.7.8/lib/ruby/
gems/shared/gems/torquebox-server-3.0.1-java/jboss/modules -jaxpmodule
javax.xml.jaxp-provider org.jboss.as.standalone -Djboss.home.dir=/root/.
rbenv/versions/jruby-1.7.8/lib/ruby/gems/shared/gems/torquebox-server-
3.0.1-java/jboss -Djboss.server.base.dir=/root/.rbenv/versions/jruby-1.7.8/
lib/ruby/gems/shared/gems/torquebox-server-3.0.1-java/jboss/standalone
-Djruby.home=/root/.rbenv/versions/jruby-1.7.8 --server-config=standalone.
xml -b 0.0.0.0
```

如果没有发现 Java/TorqueBox 进程运行，则启动它：

```
torquebox deploy --env production
torquebox run --bind-address=0.0.0.0
```

- 如果有超过一个 Razor 进程运行，将其全部杀死并再次启动 TorqueBox。
- 检查日志：是否有明显的错误？
- OS 镜像文件能够访问？检查权限和所有权。
- 映像能否解压到文件夹？再次检查 Razor 进程有无访问 ISO 解压位置的权限，该 ISO 通常在 razor/repo-store 下。在 Puppet 企业版上，默认位置是 /opt/puppet/var/razor/repo-store。
- DHCP 是否正常工作？ Razor 网络上必须运行 DHCP，网络上应该只有一个 DHCP 服务。
- DNS 是否正常工作？如果 DNS 配置不当，Chef 代理无法将节点移交给 Chef 服务器。
- 数据库是否正常工作，是否有合适的用户和权限？检查数据库连接性。如果数据库没有正确配置，无法执行任何 Razor 客户端命令；至少，检测这一点很简单。
- 确保客户端指向合适的 Razor 服务器。

如果 Razor 正常工作，但是节点没有正常安装，检查节点 /tmp 文件夹中创建的文件。该目录中的日志包含安装信息和代理移交日志。

16.6 小结

Razor 和配置工具相结合，可以形成非常强大的全自动服务器配给解决方案。使用这种部署系统，自定义环境的安装可以和服务器上线一样快。

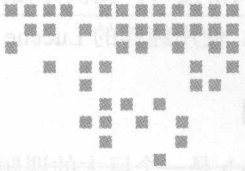
参考文献

[1] Puppet Enterprise Razor documentation: http://docs.puppetlabs.com/pe/3.2/razor_intro.html

[2] Puppet Labs Razor-Server wiki: https://github.com/puppetlabs/razor-server/wiki

[3] Puppet Labs Razor client project: https://github.com/puppetlabs/razor-client

[4] Puppet Labs Razor microkernel project: https://github.com/puppetlabs/razor-el-mk



第 17 章 Chapter 17

ELK——Elasticsearch、Logstash 和 Kibana 简介

Elasticsearch、Logstash 和 Kibana（亦称 ELK 栈）是 3 个强大的工具。Elasticsearch 是一个搜索服务器，保存数据并为索引进行优化。Logstash 是一个数据（日志）发送和清除工具。Kibana 是用于查看和分析数据的前端。它们都可以用作单独的工具，但是结合使用能够形成完美的日志管理组合。

本章简单介绍每种工具并解释如何组合它们进行有效的日志管理。本章包含如下主题：

- 理解 Elasticsearch 索引
- 使用 Elasticsearch 数据
- 安装 Elasticsearch 插件
- 使用 Elasticsearch 客户端
- 配置 Logstash 输入
- 在 Logstash 中应用过滤器
- 理解 Logstash 输出
- Kibana 中的共享和保存

17.1 Elasticsearch 概述

Elasticsearch 是 DevOps 工具链中最为出色的工具之一。越了解它，就会找到越多的用途。Elasticsearch 是一个强大的索引和搜索引擎，还提供了内建的水平可伸缩性和弹性，它

有一个 REST 风格接口，提供方便的数据输入和读取。Elasticsearch 基于强大的开放源码搜索引擎 Lucene。使用纯粹的 Lucene 有一定的挑战性，但是 Elasticsearch 却易于入门。

17.1.1 入门

Elasticsearch 是一个巨大的课题，深入地探索需要好几本书。但是，Elasticsearch 的入门很容易；在 Elasticsearch 网站上可以找到安装和设置的文档：

使用 Puppet 安装和配置：https://github.com/elasticsearch/puppet-elasticsearch。Chef 烹调书：https://github.com/elasticsearch/cookbook-elasticsearch。其他：http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/_installation.html。

一旦 Elasticsearch 运行，使用 curl 命令测试，如程序清单 17-1 所示。

程序清单 17-1 测试 Elasticsearch 状态

```
curl 'http://localhost:9200/?pretty'
{
  "status" : 200,
  "name" : "Blacklash",
  "version" : {
    "number" : "1.1.1",
    "build_hash" : "f1585f096d3f3985e73456debdc1a0745f512bbc",
    "build_timestamp" : "2014-04-16T14:27:12Z",
    "build_snapshot" : false,
    "lucene_version" : "4.7"
  },
  "tagline" : "You Know, for Search"
}
```

程序清单 17-1 中的命令显示群集状态和其他一些信息。虽然可以通过 REST 调用获得 Elasticsearch 群集健康和状态方面的详情，但是有一些出色的插件可以通过图形用户界面提供相同的信息。本章稍后将介绍插件。

17.1.2 理解索引

Elasticsearch 是一个复杂系统，对用户掩盖了其复杂性。最复杂的部分之一是 Elasticsearch 的核心——索引。如果查看数据文件夹，你就会注意到 Elasticsearch 索引是一组特殊文件。不要试图对这些文件进行直接操作；而要将它们视为数据库。当新数据加入时它们的尺寸会增长，在数据索引且索引优化（压缩）时尺寸会缩小。进行大量索引时，你可能会注意到索引文件显著增长，只是在大小上缩小到 1/3 或者 1/2。这是正常的，如果想要优化索引，还可以进行一些设置。

程序清单 17-2 中，用 curl 命令创建了名为 devops 的新索引。

程序清单 17-2　创建新索引

```
$ curl -XPUT 'localhost:9200/devops?pretty'
{
  "acknowledged" : true
}
```

程序清单 17-3 中的命令列出群集中的所有索引，验证索引创建。

程序清单 17-3　列出 Elasticsearch 索引

```
$ curl 'localhost:9200/_cat/indices?v'
health index               pri rep docs.count docs.deleted store.size
yellow devops                5   1          0            0       495b
yellow _river                1   1          3            0        10kb
yellow mybooks               5   1        460            0    244.7mb
yellow .marvel-2014.05.02    1   1        765            0      2.4mb
```

索引可能有不同的状态和多个分片。每个 Elasticsearch 群集可能有多个索引，索引的数量取决于数据的结构。

17.1.3　使用数据

理解 Elasticsearch 可以索引的数据很重要。你想要索引的数据将决定文档的格式，文档是保存在 Elasticsearch 中的 JSON 对象，可以将其视为 nosql 数据库中的一个条目。为了使数据的检索更容易，保存在索引中的文档应该有一致的格式。这种一致性和格式取决于最终用户和保存的数据，正像在 nosql 数据库中那样。文档格式或者 JSON 可以根据需要选择，只要它是有效的 JSON 格式。每个文档由 0 个或者多个字段（键 – 值对）组成。每对值可能简单也可能复杂，数据保存到索引之后，在内部处理（索引）为可搜索数据。

要在 Elasticsearch 中存储数据，需要使用其 API。Elasticsearch API 简单易用，程序清单 17-4 展示了添加简单条目的例子以及响应。

程序清单 17-4　在索引中添加数据

```
curl -XPUT 'localhost:9200/devops/chef/1?pretty' -d '
> {
>   "recipes": "vsphere"
> }'
{
  "_index" : "devops",
  "_type" : "chef",
  "_id" : "1",
  "_version" : 1,
  "created" : true
}
```

在这个例子中，我们在 devops 索引中添加 chef 类型的数据，ID 为 1。实际数据是键 - 值对“recipes: vsphere”。

存放在 Elasticsearch 中的数据可以遵循用户定义的模式，也可以采用自由形式。读取数据时，结果将以完整文档的形式返回。程序清单 17-5 展示了相同文档读回的情况。

程序清单 17-5 列出 Elasticsearch 索引

```
$ curl -XGET 'localhost:9200/devops/chef/1?pretty'
{
  "_index" : "devops",
  "_type" : "chef",
  "_id" : "1",
  "_version" : 1,
  "found" : true, "_source" :
{
  "recipes": "vsphere"
}
}
```

现在一切都还很简单，但是数据如何搜索呢？最简单的搜索没有任何参数：

```
$ curl -XGET 'localhost:9200/devops/_search'
```

冗长的结果将在一行上显示，使用命令行文本格式化工具或者 http://jsbeautifier.org/ 等浏览器工具可以得到格式化的 JSON。程序清单 17-6 显示了搜索中的格式化结果。

程序清单 17-6 格式化的搜索结果

```
{
    "took": 1,
    "timed_out": false,
    "_shards": {
        "total": 5,
        "successful": 5,
        "failed": 0
    },
    "hits": {
        "total": 2,
        "max_score": 1.0,
        "hits": [{
            "_index": "devops",
            "_type": "chef",
            "_id": "1",
            "_score": 1.0,
            "_source": {
                "recipes": "vsphere"
            }
```

```
        }, {
            "_index": "devops",
            "_type": "puppet",
            "_id": "1",
            "_score": 1.0,
            "_source": {
                "manifest": "jenkins"
            }
        }]
    }
}
```

不要执行从索引中返回所有数据的搜索，除非你打算进行调试且没有其他选择。如果想要检查所有数据，有些插件可以在 Web 浏览器中检查所有字段和数据。

```
$ curl -XGET 'http://localhost:9200/devops/_search?q=jenkins'
```

上述命令显示了一个简单查询，在 devops 中搜索 jenkins 的所有实例，这将返回在任意字段中包含 jenkins 的所有文档，如程序清单 17-7 所示。

程序清单 17-7　搜索 jenkins 的格式化搜索结果

```
{
    "took": 1,
    "timed_out": false,
    "_shards": {
        "total": 5,
        "successful": 5,
        "failed": 0
    },
    "hits": {
        "total": 1,
        "max_score": 1.0,
        "hits": [{
            "_index": "devops",
            "_type": "puppet",
            "_id": "1",
            "_score": 1.0,
            "_source": {
                "manifest": "jenkins"
            }
        }]
    }
}
```

Elasticsearch 查询语言很复杂，使用 curl 或者其他直接应用程序编程接口（API）调用，

也需要使用 JSON 格式。匹配包含 jenkins 的任意文档的简单查询如程序清单 17-8 所示。

程序清单 17-8 搜索 jenkins 的 JSON 查询示例

```
{
    "query": {
        "bool": {
            "must": [],
            "must_not": [],
            "should": [{
                "query_string": {
                    "default_field": "_all",
                    "query": "jenkins"
                }
            }]
        }
    },
    "from": 0,
    "size": 10,
    "sort": [],
    "facets": {}
}
```

你很有可能对使用 curl 提交复杂查询不感兴趣。更好的查询方式是使用脚本语言客户端或者与应用的直接代码集成。插件在查询纠错或者进行一次性搜索时也很有用。

关于 Elasticsearch 查询语言的更多知识参见 http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/query-dsl-query-string-query.html。

17.1.4 安装插件

Elasticsearch 自带多种插件，帮助数据输入和分析，这样就不需要手工完成所有任务。可以这样安装大部分插件：

```
$ bin/plugin --install mobz/elasticsearch-head
```

Elasticsearch-head 是我最喜欢的插件之一，也是设置完 Elasticsearch 之后首先安装的。这个插件是一个简单的 Web 前端，提供群集概况、检查数据和执行搜索。

图 17-1 展示了 elasticsearch-head 插件中的群集概况。

简单的界面下有许多数据。Cluster Overview（群集概况）选项卡显示群集的总体状态，每个下拉菜单可以显示更多细节。

Browser（浏览器）选项卡（参见图 17-2）检查 Elasticsearch 索引的所有数据项，可以通过索引、类型和字段缩小范围。Structured Query（结构化查询）和 Any Request（任何请求）选项卡提供查询语言包装器，以及对 Elasticsearch 群集发起任何请求的一种手段。

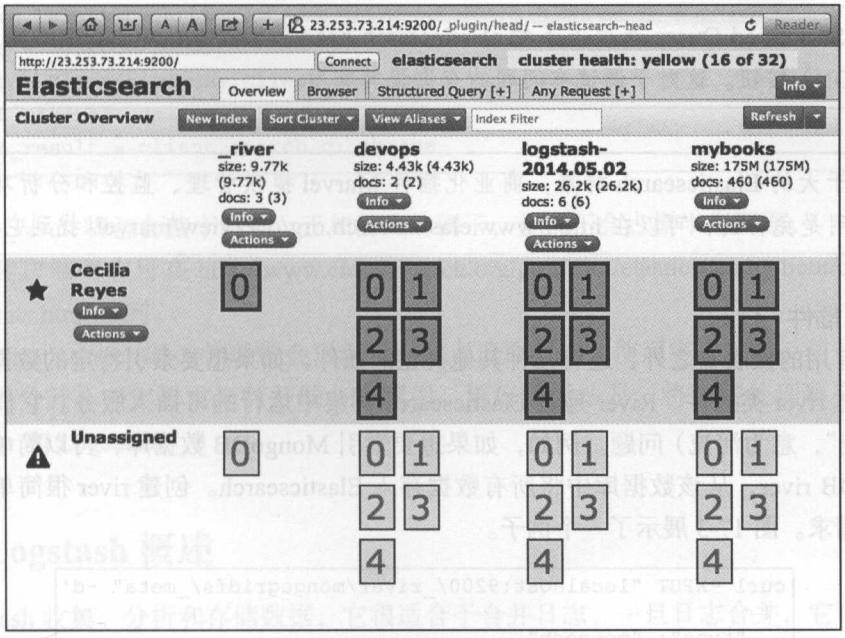


图 17-1　Elasticsearch 群集

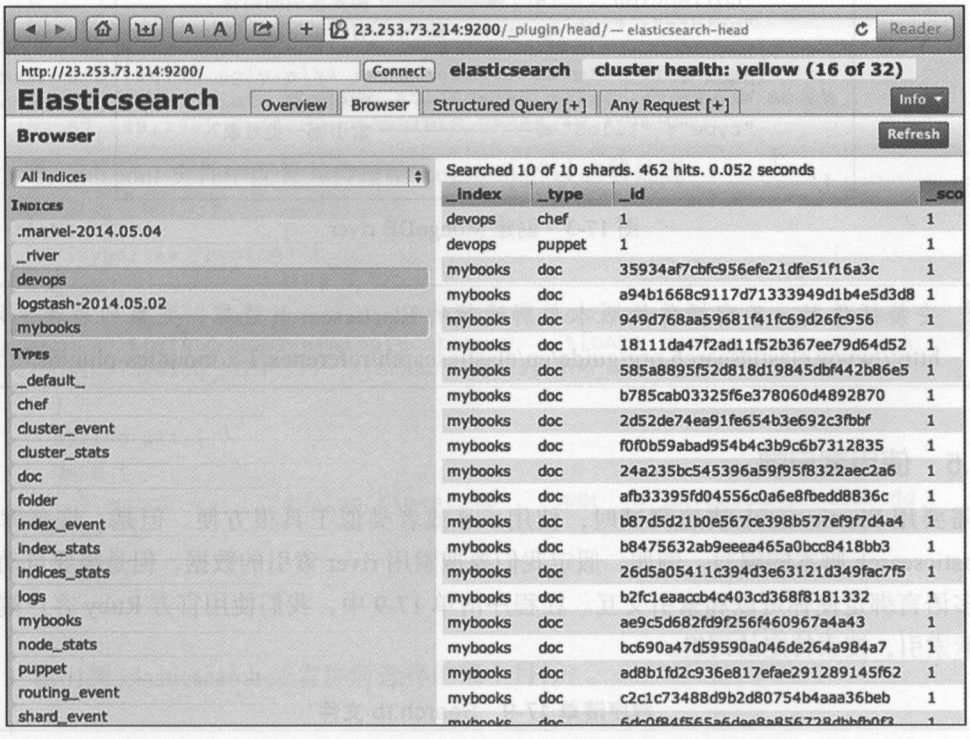


图 17-2　Elasticsearch 索引详情和单独文档

提示 在 Structured Query 选项卡下的 elasticsearch-head 界面是 Show Raw JSON（显示原始 JSON）按钮。这对于调试有问题的查询极其有用。

提示 对于大的 Elasticsearch 群集，商业化插件 Marvel 提供管理、监控和分析功能。开发使用是免费的，可以在 http://www.elasticsearch.org/overview/marvel/ 找到它。

River 插件

除了有用的仪表盘之外，还有几种其他类型的插件。如果想要索引特定的数据源，可能有必要关注 river 类插件。River 是在 Elasticsearch 群集中运行的可插入服务。它们解决数据流（“river”，意为河流）问题。例如，如果想要索引 MongoDB 数据库，可以简单地配置一个 MongoDB river，从该数据库中将所有数据存入 Elasticsearch。创建 river 很简单，只需要发出 curl 请求。图 17-3 展示了一个例子。

```
curl -XPUT "localhost:9200/_river/mongogridfs/_meta" -d'
 {
   "type": "mongodb",
     "mongodb": {
       "db": "my_database",          <-- 数据库名
       "collection": "fs",           <-- 需要索引的集合
       "gridfs": true                <-- 有GridFS对象吗?
     },
     "index": {
       "name": "gutenberg",          <-- 索引名称
       "type": "files"               <-- 索引哪一类对象?
     }
 }'
```

图 17-3 创建 MongoDB river

提示 安装插件时，确定插件的版本与所运行的 Elasticsearch 兼容。完整列表参见网站：http://www.elasticsearch.org/guide/en/elasticsearch/reference/1.x/modules-plugins.html。

17.1.5 使用客户端

需要用 Elasticsearch 快速测试时，使用 curl 或者类似工具很方便。但是，许多库简化了 Elasticsearch 脚本的编写。例如，假定我们要搜索用 river 索引的数据，但是需要自动化操作。多语言绑定使你可以和索引交互。在程序清单 17-9 中，我们使用官方 Ruby 客户端连接到搜索索引，搜索特定的词组。

程序清单 17-9 Search.rb 文件

```
require 'rubygems'
```

```
require 'elasticsearch'
phrase = "custom phrase"
client = Elasticsearch::Client.new host: 'localhost'
client.cluster.health
search_result = client.search q: phrase
```

search_result 将包含整个结果，正如前面的例子，查询不会占据半个页面。

当前客户端列表可在 http://www.elasticsearch.org/guide/en/elasticsearch/client/community/current/clients.html 找到。

使用 Elasticsearch 时，你通常会更新索引或者读取数据。使用客户端时，将写入索引的任务和读取分开，一定要重新打开同一个索引。更好的做法是，将某些处理移交给其他工具，如 river 和 Logstash。

17.2　Logstash 概述

Logstash 收集、分析和存储数据，它很适合于合并日志，一旦日志合并，它可以提供方便的存取，而无须访问各种不同的系统。

Logstash 的工作方式可以 3 个简洁的步骤描述：

1. 输入数据，如 auth.log 文件中的内容：

```
May  5 01:46:06 egle-elk1 sshd[16057]: reverse mapping
checking getaddrinfo for 202.57.189.199.client.siamtech.ac.th
[202.57.189.199] failed - POSSIBLE BREAK-IN ATTEMPT!
```

2. 在 auth.conf 文件中配置并传递给 Logstash 的过滤器处理：

```
filter {
  if [type] == "syslog" {
    grok {
          match => { "message" => "%{SYSLOGTIMESTAMP:syslog_time-
stamp}    %{SYSLOGHOST:syslog_hostname}    %{DATA:syslog_program}(?:\
[%{POSINT:syslog_pid}\])?: %{GREEDYDATA:syslog_message}" }
    }
    syslog_pri { }
    date {
        match => [ "syslog_timestamp", "MMM  d HH:mm:ss", "MMM dd
HH:mm:ss" ]
    }
  }
}
```

3. 输出到 Elasticsearch 或者你所选择的某个目标：

```
{
       "message" => "May  5 01:57:13 egle-elk1 sshd[16642]: reverse
```

```
mapping checking getaddrinfo for 202.57.189.199.client.siamtech.ac.th
[202.57.189.199] failed - POSSIBLE BREAK-IN ATTEMPT!",
      "@version" => "1",
    "@timestamp" => "2014-05-05T01:57:14.102Z",
          "host" => "egle-elk1",
          "path" => "/var/log/auth.log"
}
```

第 1 步中的内容是你已经拥有的。第 2 步可以在一个 Logstash 食谱中找到，第 3 步是 Logstash 生成的。我们将在本节中介绍所有步骤。

17.2.1 入门

首先，从 http://logstash.net/ 下载 Logstash。Logstash 是一个 Java 应用程序，所以需要 JVM，遵循网站上的安装指南：http://logstash.net/docs/1.4.0/tutorials/getting-started-with-logstash。

此外，你需要决定使用什么存储。如果想要让 Logstash 和 Elasticsearch 一起使用，验证 Elasticsearch 运行，就为日志处理做好了准备。在两者都运行时，需要创建一个配置文件，包含输入、过滤器和输出配置。

17.2.2 配置 Logstash 输入

Logstash 处理日志，从其角度看，日志可以定义为具有时间戳的任何内容。将数据输入 Logstash 有多种方法；默认方法是使用 Logstash 代理。除了使用 Logstash，还可以配置更适合环境的替代方法。其他可用日志传输方法可以参考 Logstash 食谱：http://cookbook.logstash.net/recipes/log-shippers/。

使用配置文件中的 input 段配置 Logstash 输入，可用的输入列表正在不断增长；如果在 http://logstash.net/docs/1.4.0/ 的文档中没有看到首选的输入，可以添加自己的输入类型。我们将在例子中使用 Logstash 烹调书（http://cookbook.logstash.net/）中找到的一个食谱（这些食谱不应该和 Chef 食谱混同）。该例子用于 syslog 日志处理。首先，我们创建如程序清单 17-10 中所示的 syslog.conf 文件。

程序清单 17-10 syslog.conf 文件

```
input {
  tcp {
    port => 5000
    type => syslog
  }
  udp {
    port => 5000
    type => syslog
  }
}
```

```
filter {
  if [type] == "syslog" {
    grok {
      match => { "message" => "%{SYSLOGTIMESTAMP:syslog_timestamp}
%{SYSLOGHOST:syslog_hostname} %{DATA:syslog_program}(?:\[%{POSINT:syslog_
pid}\])?: %{GREEDYDATA:syslog_message}" }
      add_field => [ "received_at", "%{@timestamp}" ]
      add_field => [ "received_from", "%{host}" ]
    }
    syslog_pri { }
    date {
      match => [ "syslog_timestamp", "MMM  d HH:mm:ss", "MMM dd HH:mm:ss" ]
    }
  }
}

output {
  elasticsearch { host => localhost }
  stdout { codec => rubydebug }
}
```

在例子中，输入是 syslog，输出为 Elasticsearch 和标准输出。创建配置文件之后，启动 Logstash：

```
$ bin/logstash -f syslog.conf
```

因为我们启用了控制面板日志，所以可以在控制面板中看到处理过的日志。

程序清单 17-11 展示了保存在 Elasticsearch 中供读取的文档格式。

程序清单 17-11　控制面板日志输出

```
{
                 "message" => "\r",
                "@version" => "1",
              "@timestamp" => "2014-05-02T11:26:19.709Z",
                    "host" => "127.0.0.1:45058",
                    "type" => "syslog",
                    "tags" => [
        [0] "_grokparsefailure"
    ],
    "syslog_severity_code" => 5,
    "syslog_facility_code" => 1,
      "syslog_facility" => "user-level",
      "syslog_severity" => "notice"
}
```

17.2.3 应用过滤器

你可能已经注意到，程序清单 17-10 中的配置文件包含一个 filter 段。这是告诉 Logstash 如何解析输入数据的部分。过滤器按照列出的顺序应用，可以包含条件和插件。为了节约编写正则表达式的时间，Logstash 提供了 Grok 过滤器。

Grok

Grok 是一个用于解析日志数据的特殊过滤器，有了它，用户就不必编写正则表达式，而且它是解析非结构化数据的最佳途径。Grok 内建了 100 多种模式，可以解析最复杂的数据。在该项目的 GitHub 页面可以看到可用的模式：https://github.com/elasticsearch/logstash/tree/master/patterns，也可以添加自己的模式。

如果想添加自己的模式或者 Grok 表达式有缺陷，Grok Debugger 是一个方便的工具：http://grokdebug.herokuapp.com/。Grok Debugger 不仅能够帮助你调试自己的模式，还可以根据输入生成模式，而且可以方便地查看所有可用模式。

数据转换

在过滤器的帮助下，在日志处理时可以转换数据、添加额外的字段。所以，如果原始数据使用简单的格式（如程序清单 17-12），可以转换、删除和添加附加信息。

程序清单 17-12 未格式化的日志文件条目

```
May  5 01:46:06 egle-elk1 sshd[16057]: reverse mapping checking getaddrinfo
for 202.57.189.199.client.siamtech.ac.th [202.57.189.199] failed - POSSIBLE
BREAK-IN ATTEMPT!
```

通过过滤器处理之后，我们可以添加附加信息，如程序清单 17-13 所示。

程序清单 17-13 添加字段的过滤器

```
add_field => [ "received_at", "%{@timestamp}" ]
add_field => [ "received_from", "%{host}" ]
```

删除字段的过滤器如下：

```
remove_field => [ "syslog_hostname", "syslog_message", "syslog_timestamp" ]
```

修改字段的过滤器如下：

```
replace => [ "@source_host", "%{syslog_hostname}" ]
```

可以对数据进行哪些类型的处理？参见每个过滤器的文档。例如，程序清单 17-14 包含了 Grok 的所有可用过滤器。

程序清单 17-14 Search.rb 文件

```
grok {
    add_field => ... # hash (optional), default: {}
```

```
    add_tag => ... # array (optional), default: []
    break_on_match => ... # boolean (optional), default: true
    drop_if_match => ... # boolean (optional), default: false
    keep_empty_captures => ... # boolean (optional), default: false
    match => ... # hash (optional), default: {}
    named_captures_only => ... # boolean (optional), default: true
    overwrite => ... # array (optional), default: []
    patterns_dir => ... # array (optional), default: []
    remove_field => ... # array (optional), default: []
    remove_tag => ... # array (optional), default: []
    tag_on_failure => ... # array (optional), default: ["_grokparsefailure"]
}
```

操纵数据之后，可以将其导向多种输出。

17.2.4　理解输出

输出是 Logstash 存储处理后日志数据的地方。可以同时指定一个以上的输出。Logstash 的首选目标是 Elasticsearch，尤其是在希望使用 Kibana 进行数据分析时，配置 Elasticsearch 作为输出的更多说明参见 http://logstash.net/docs/1.4.0/outputs/elasticsearch 上的文档。但是，最简单的设置中只需要主机信息。

除了将输出发送到 Elasticsearch 之外，还有多种其他目标。实际上，Logstash 可以一次将数据发送给不止一个目标。将输出导向多个目标的方法参见程序清单 17-15。

程序清单 17-15　多重目标

```
output {
  elasticsearch { host => localhost }
  stdout { codec => rubydebug }
}
```

除了 Elasticsearch 和标准输出之外，其他一些目标包括电子邮件、文件、pagerduty 和 jira。所以，你可以过滤日志，找出具体的错误，并通过传呼机、电子邮件和 jira 问题通知用户。对于测试，也可以不输出。

Logstash 处理数据并保存到 Elasticsearch 之后，可以在 Kibana Logstash 仪表盘中可视化和分析。

17.3　Kibana 概述

Kibana 是一个仪表盘，用于可视化具有时间戳的任何数据。它用 HTML 和 JavaScript 编

写，唯一的要求是有 Web 服务器和一行配置。只要下载 Kinaba，解压文件并拖放到 Web 服务器，就可以运行 Kibana。如果同一服务器上没有运行 Elasticsearch，则修改 config.js 文件，指向 Elasticsearch 群集。

第一次在 Web 浏览器中打开 Kibana 之后，会看到一个欢迎屏幕，屏幕上有指向几个默认仪表盘的链接（参见图 17-4）。

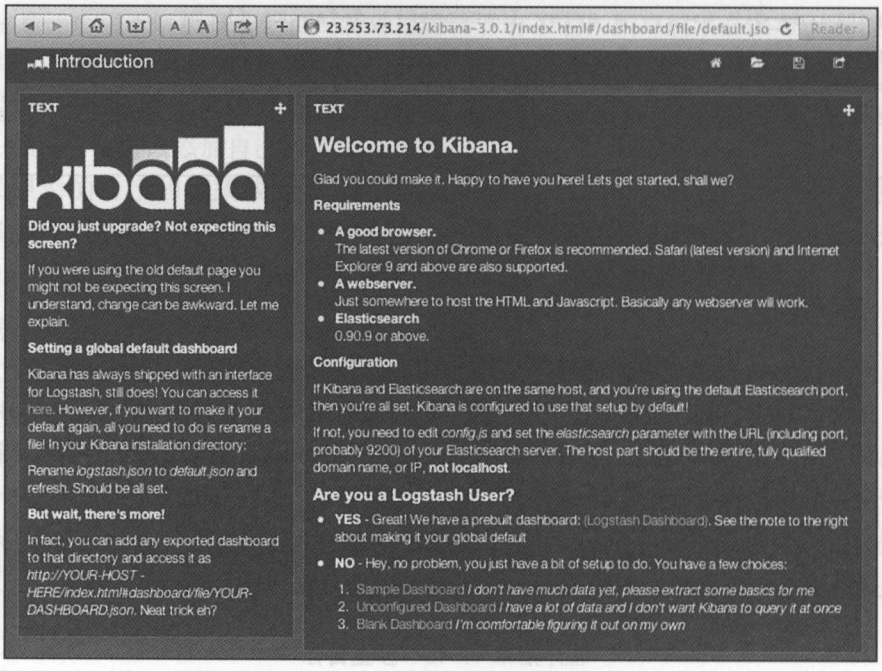


图 17-4 Kibana 欢迎屏幕

你可以构建自己的仪表盘，但是我们打算使用 Logstash 仪表盘。打开 Logstash 仪表盘之后，会看到数据自动加载，显示多个用于自定义仪表盘的齿轮和按钮。为了将深色的仪表盘更改为浅色，找到右上角的 Configure Dashboard（配置仪表盘）齿轮图标并单击，之后会显示仪表盘设置窗口，有多个编辑仪表盘设置的选项，如图 17-5 所示。

在图 17-5 中显示了更改样式的选项。我们打算切换到浅色。除了样式之外，还可以调整 Kibana 指向的 Elasticsearch 索引，以及 Rows、Controls 和 Timepicker 选项下的其他显示选项。

单击任何图形都可以放大以观察更多细节。对于柱状图，可以在可疑的活动上选择时间跨度研究峰值。图 17-6 展示了一个例子。

在 Events over Time 仪表盘下有对应的 All Events（全部事件）表格。这个表格包含默认显示的所有字段。可以仅选择所需的字段并修改其顺序，调整视图。图 17-7 展示了事件表格中仅显示主机和消息字段的例子。

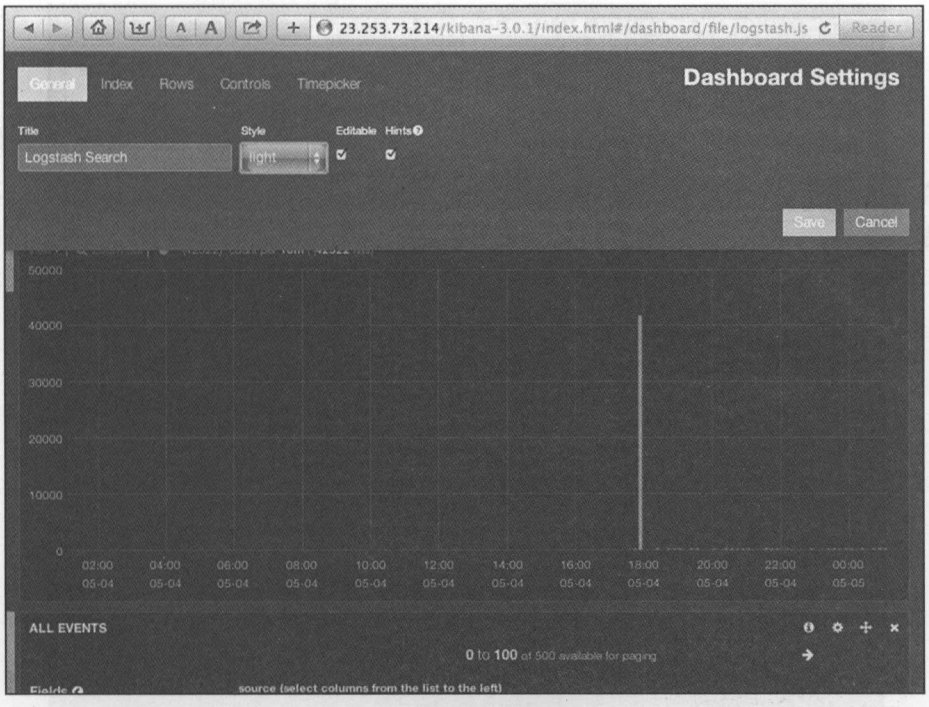


图 17-5　将 Kibana 样式更改为浅色主题

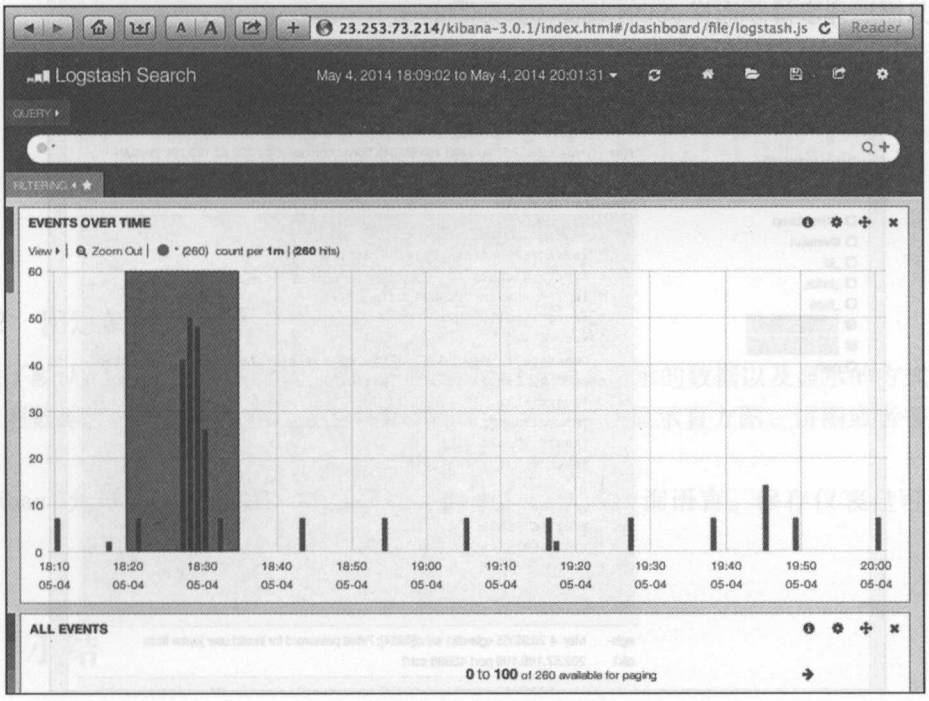


图 17-6　放大日期范围

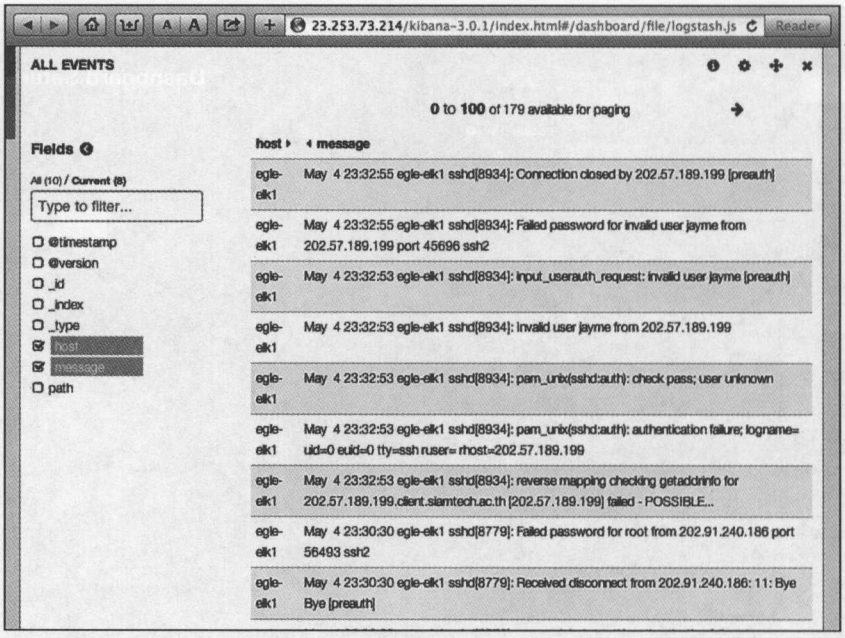


图 17-7 自定义事件表格

每个事件也可以点击，并可使用多种格式。要查看保存在 Elasticsearch 中的 JSON 格式事件，只需展开事件并选择 JSON 视图（参见图 17-8）。

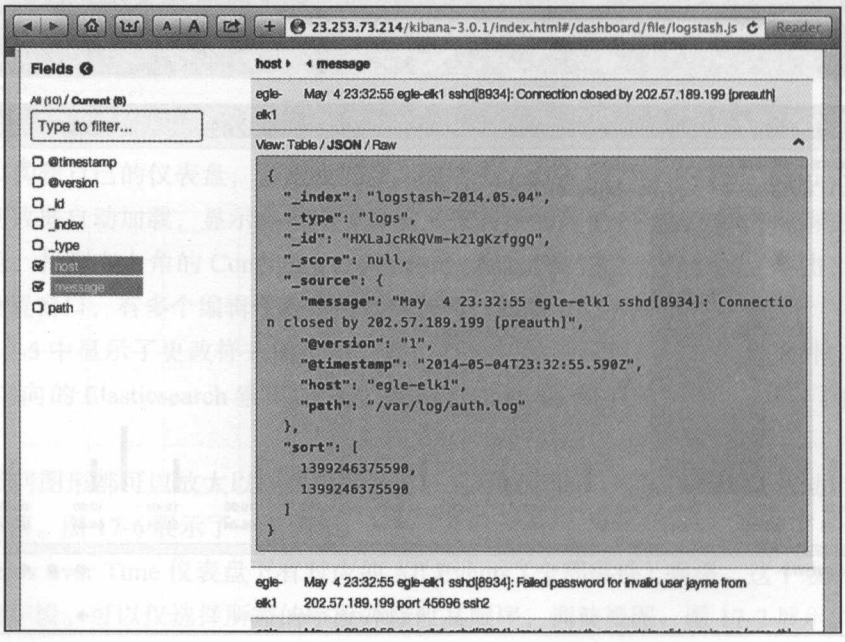


图 17-8 事件详情 JSON 视图

所有不同的视图可以共享、保存和再次查看。如果没有保存当前视图，不要刷新浏览器。

17.3.1　共享和保存

不同控制面板可以共享，如果想要再次加载就必须保存。共享和保存选项都在右上角。保存的选项可以通过单击保存图标旁边的文件夹图标加载。如图 17-9 所示，共享仪表盘时，会向你提供指向刚刚创建的同一个仪表盘的 URL。

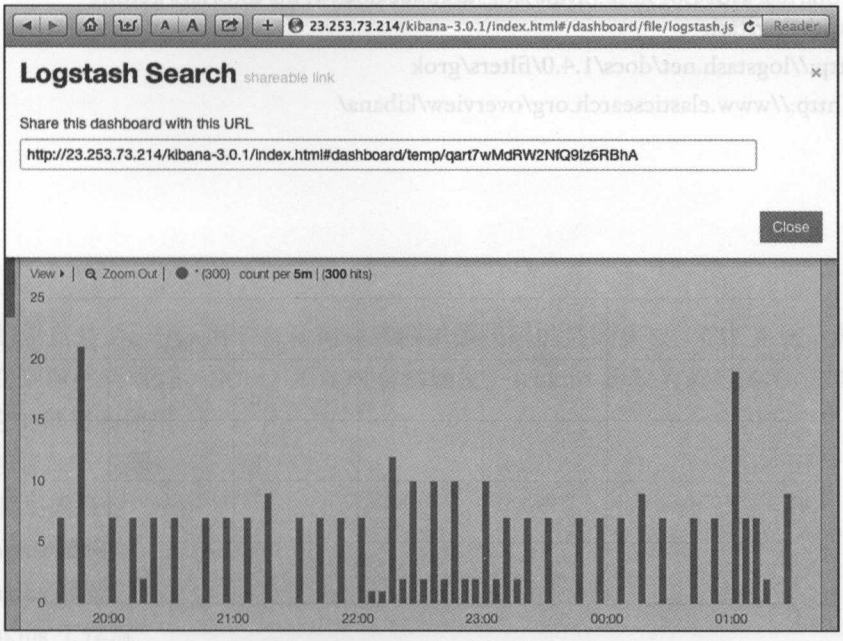


图 17-9　共享 Kibana 临时视图

17.3.2　自定义数据视图

除了默认的 Logstash 仪表盘，Kibana 允许用户自定义显示的数据以及显示的方式。如果你有地理数据，可以尝试添加一个地图面板。其他面板包括显示直方图、饼图或者普通文本的手段。

Kibana 允许用户通过 GUI 探索日志，而不需要学习查询语言。保存仪表盘可以代替搜索。

17.4　小结

Elasticsearch、Logstash 和 Kibana 是可以独立使用的强大工具。但是，当它们组合使用

时，能够真正发挥威力，组成一流的日志管理套件。它们只需要简单的设置就可以处理许多数据并进行可视化。整个 ELK 栈都可以通过 Puppet 清单或者使用 Chef 食谱安装。

参考文献

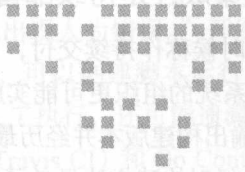
[1] Elasticsearch documentation: http://www.elasticsearch.org/guide/

[2] Logstash: http://logstash.net/

[3] Getting Started with Logstash: http://logstash.net/docs/1.4.0/tutorials/getting-started-with-logstash

[4] Grok: http://logstash.net/docs/1.4.0/filters/grok

[5] Kibana: http://www.elasticsearch.org/overview/kibana/



第 18 章 Chapter 18

用 Jenkins 实现持续集成

本章的焦点是 DevOps 用于自动构建和测试源代码的持续集成（CI）系统。构建过程在新更改提交到源代码管理（SCM）系统时自动触发。Jenkins 是较为流行的 CI 工具之一，我们将研究它的部署和使用。

本章包含如下主题：

- 持续集成概念
- Jenkins 架构
- Jenkins 部署
- Jenkins 工作流
- 质量保证团队

18.1 持续集成概念

快速基础设施部署、自动配置关键服务是很好的，而快速部署无缺陷、面向客户的应用程序对业务成功更为关键。

传统软件开发团队在编写代码和产品交付之间有多个步骤，包括源代码提交、由源代码构建二进制程序、质量保证（QA）测试、预演和最后的生产部署。这些步骤往往包含人工过程和移交，可能在源代码中现有的问题之上引入其他缺陷。如果有一种方法能在源代码部署过程中增加自动化成分，不是很好吗？这就是 CI 系统的切入点。

CI 系统允许定义从源代码开发开始，到经过测试及验证的代码自动为生产部署进行试运行为止的管道。前面列出的中间步骤（包括测试）通过 CI 系统自动执行。

18.1.1 持续集成还是持续部署

因为 CI 有时候称作持续交付，可能导致新的 DevOps 从业人员将其与持续部署混同。刚刚开始入手 CI 系统的组织更可能实施持续集成，这可以视为持续预演。

利用 CI，输出构建版本并经历最终验收测试，然后人工部署到生产环境中。在 CI 组织中，专业人员可能已经对构建方法具备了信心，新的构建可以自动投产。这就是持续部署（CD）。

不管采用的是 CI 还是 CD 方法。每种工作流所使用的工具集相同。所以，在本章中将 Jenkins 及类似的工具统称为 CI/CD。

18.1.2 测试自动化

应用程序问题的根源可能很多，但是通常可以归为几类：编译问题、功能缺陷和逻辑缺陷。“罪责”可能是操作系统（OS）配置更改、开发人员引入缺陷或者第三方依赖性的变化，这三者中，第二种很常见。

为什么开发人员引入这些问题的风险更大？对于开发人员来说，分布到不同的敏捷团队、物理位置、在不同时间表下工作以及处于不同时区是常见的做法。事实上，这些开发人员有着不同的优先级以及专业水平，进而造成了上述的问题。尽管这似乎是难以克服的障碍，但是解决方案很简单：经常测试。根据应用程序的编程语言，有多种工具可以显著地降低软件开发过程中的风险。

第一步是让开发人员在软件开发过程之中（甚至之前）思考测试案例。软件行业对于测试驱动开发（TDD）的优点有争议，但是在软件开发过程中认真地思考代码的预期行为，是无可争议的。

开发人员可以用一些行业标准工具创建单元测试，验证代码中的功能性部分。此类工具包括用于 Java 的 JUnit、用于 Python 的 PyUnit、用于 Ruby-on-Rails 的 RSpec 和 TestUnit 以及用于 Microsoft .NET 的 NUnit。这些工具抽象了测试时需要的重复工作（检查有效返回值、逻辑等），方便了测试的快速编写。它们可以快速反馈，因为在任何测试失败时，执行会立刻停止。

单元测试的思路是，这些测试应该覆盖应用程序的大部分功能。随着开发的继续，这些测试可以确保：

1. 新功能正常工作。
2. 现有功能继续工作。
3. 以前解决的缺陷不会重新引入（回归测试）。

开发人员在软件开发过程中反复测试，确保所有测试在源代码提交之前通过。

构建 Web 应用时，有些功能只能通过浏览器测试。这些应用程序可以使用 Selenium webdriver 和 Capybara 等工具测试，上述工具可与前面提到的测试框架集成，它们所包含的库使你可以创建运行浏览器会话、打开应用程序的不同页面并验证生成页面上内容的程序。

在开发人员对用户界面（UI）作出调整时，可以运行这些测试验证内容仍然正确显示。

你可能会这样想，“这听起来是人工进行的，有可能出现人为错误。”开发人员可能没有意识到其他测试依赖于她的代码。对于较小的应用程序，可以创建脚本运行所有单元测试，但是可能很费时，从而使开发人员无法经常测试。如果人工执行，还可能遗漏某些测试。

幸运的是，Jenkins、Travis Continuous Integration（Travis CI）和 Go Continuous Delivery（GoCD）等 CI/CD 工具可以创建由每个单元测试的构建步骤组成的管道。在源代码构建为打包的二进制程序并投入预演之前，必须通过 CI/CD 管道中定义的每个单元测试（参见图 18-1）。

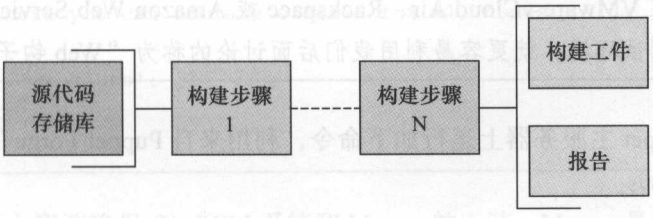


图 18-1　CI/CD 构建工作序列

CI/CD 系统可以轮询 SCM 系统获得更改情况，更常见的是 SCM 系统在每次源代码提交之后触发 CI/CD 系统，生成新的构建版本。在构建结束时生成报告。如果没有错误，编译后的代码（称作构建工件）将保存在 CI/CD 管理员规定的位置。测试工具和 CI/CD 系统都有助于形成高效的 DevOps 环境。有缺陷的代码不太可能部署，因为错误在开发过程的较早期就被找出，避免了技术负载的积累。

18.2　Jenkins 架构

Jenkins 是一个可扩展的平台，具备许多插件，可以支持没有现成支持的技术和功能。例如，尽管 Git 是流行的 SCM 系统，但 Jenkins 默认只支持 SVN 和 CVS。这没有任何问题！插件可以实现 Git 支持，帮助你使用现有的 SCM 存储库。

Jenkins 由一个主系统和一个或多个可选的、从主系统接受任务的从系统组成。主系统能够执行构建和报告任务，但是为达到可伸缩性的目的，应该考虑多机器设置，由从节点执行所有工作。图 18-2 展示了由一台 Jenkins 主服务器和两台 Jenkins 从服务器组成的架构示例。用户与主服务器交互，构建

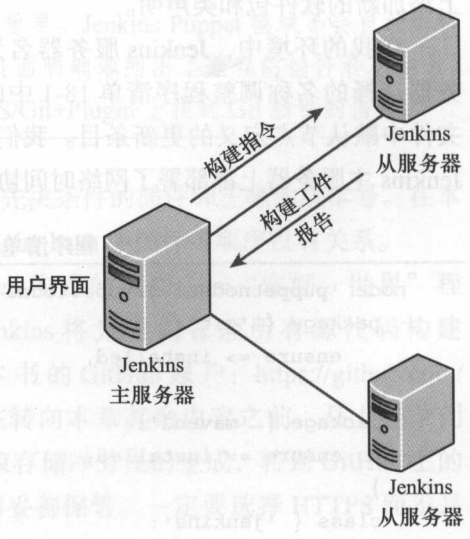


图 18-2　Jenkins 架构

指令根据分配给每个从服务器的执行者（或者并行构建过程）数量发送到从服务器。不管构建发生在主服务器还是从服务器，构建工件和报告都保存在主服务器上供用户访问。

18.3 Jenkins 部署

我们的测试环境将包含两台服务器：Puppet 主服务器，以及作为 Jenkins 主服务器、运行 Puppet 代理的 Ubuntu 服务器。

注意 如果可以在 VMware vCloud Air、Rackspace 或 Amazon Web Services 等公共云提供商部署这个测试环境，就更容易利用我们后面讨论的称为“Web 钩子”的高级功能。

我们将在 Puppet 主服务器上运行如下命令，利用来自 Puppet Forge 的一个流行 Jenkins 模块（rtyler/jenkins）：

```
puppet module install rtyler-jenkins
```

如果这是一个新环境，还要确保下载 NTP 模块，以便保持 Puppet 主服务器和代理服务器时间同步：

```
puppet module install puppetlabs-ntp
```

一旦正常部署了 Puppet 主服务器和 Puppet 代理，将程序清单 18-1 中的代码添加到 Puppet 主服务器上的 /etc/puppetlabs/puppet/environments/production/manifests/site.pp 文件。如果有本书前几章中使用过的现有 Puppet 主服务器和代理部署，也可以只在现有的代理服务器上添加新的软件包和类声明。

在我的环境中，Jenkins 服务器名为 puppetnode1.devops.local。一定要根据为 Jenkins 服务器选择的名称调整程序清单 18-1 中的节点声明。注意，我们的示例代码还包含了 site.pp 文件中默认节点定义的更新条目。我们用清单中的 node default 段，确保 Puppet 主服务器和 Jenkins 主服务器上都部署了网络时间协议（NTP）

程序清单 18-1 Jenkins 部署指令

```
node 'puppetnode01.devops.local' {
  package { 'git':
     ensure => installed,
  }
  package { 'maven2':
     ensure => installed,
  }
  class { 'jenkins':
    configure_firewall => false,
```

```
    plugin_hash => {
      'credentials' => { version => '1.9.4' },
      'ssh-credentials' => { version => '1.6.1' },
      'git-client' => { version => '1.8.0' },
      'scm-api' => { version => '0.2' },
      'git' => { version => '2.2.1' },
    }
  }
}

node default {
  # This is where you can declare classes for all nodes.
  # Example:
  #   class { 'my_class': }
  include 'ntp'
}
```

清单中的前两个资源确保 Git SCM 系统和 Maven 的安装。Maven 是我们将要用于 Java 应用程序示例的 Java 构建系统。Maven 系统的好处之一是可以自动生成构建过程的相关报告，并为 Jenkins 生成用户可用的部署包（按照 Java 术语，称作 jar 文件）。这些因素帮助我们更轻松地验证构建的成败。

接下来，我们告诉服务器如何部署 Jenkins 包。Tyler 出色地编写了一个模块，使 Jenkins 的部署和配置变得轻而易举。我们只需要设置几个选项。首先，指定 Jenkins 部署期间不需要配置防火墙。然后，包含设置正常运行所需的 Jenkins 插件散列。

注意　所要安装的 Jenkins 插件及插件版本号的细节很重要。Jenkins Puppet 模块不会自动解析插件的依赖性。幸运的是，每个插件的 Wiki 页面明确地列出了兼容的插件和版本号。可以在 https://wiki.jenkins-ci.org/display/JENKINS/Git+Plugin 上找到 Git 插件的信息。

我们想要 Git Jenkins 插件，必须确保部署了作为先决条件的插件和正确的版本号。在本书编写时，Git 插件 2.2.1 版本是最新的发行版本。plugin_hash 中插件的顺序没有关系。

我们在构建中将要使用的应用程序是由 Maven 构建的简单 Java “你好，世界” 程序。没有必要全面了解 Maven 或者 Java，因为 Jenkins 将为我们管理所有源代码构建和报告生成任务。样板应用的源代码托管在本书的 GitHub 账户：https://github.com/DevOpsForVMwareAdministrators/mvn-hello-world。在转向本章其余内容之前，马上登录到 GitHub，创建这个存储库的一个分支。当 GitHub 结束存储库分支的生成，转到 GitHub 上的存储库拷贝，将 HTTPS 克隆 UTL 复制到文本编辑器妥善保管。一定要选择 HTTPS 而不是 SSH 或者 Subversion，图 18-3 中展示了一个例子。

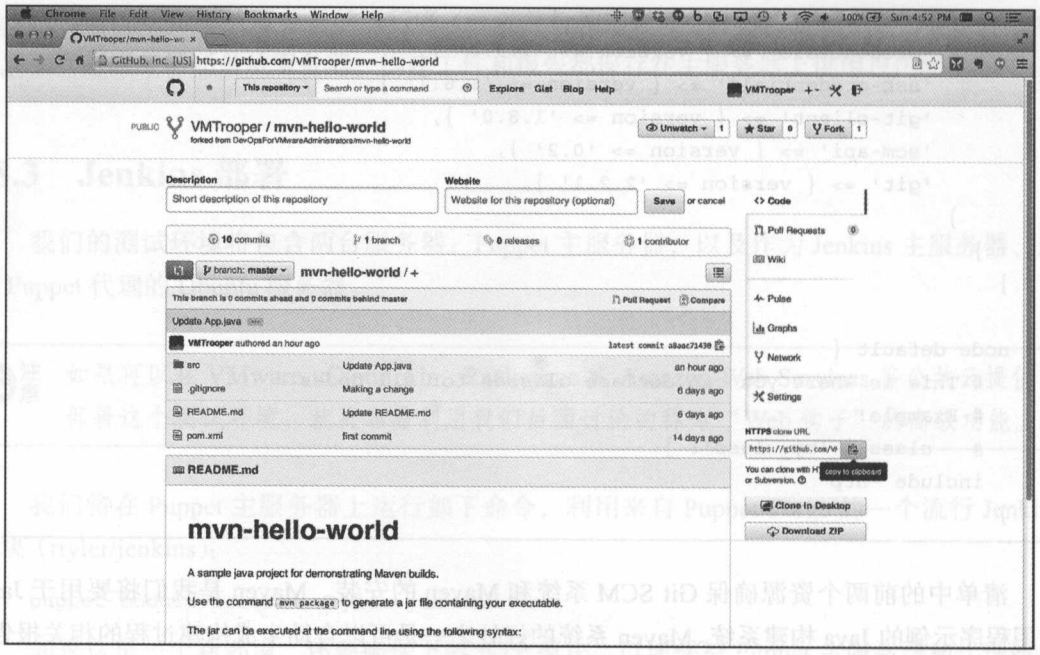


图 18-3 GitHub 存储库

18.4 Jenkins 工作流

Jenkins 服务器已经部署，我们首先注意系统先决条件，然后继续自动化应用程序构建。

Jenkins 服务器配置中有一些我们没有介绍的项目，包括用户凭据、安全性加固等。相反，我们将焦点放在构建系统正常运行必需的核心配置：

- 我们打算使用的编程语言
- 构建所用的源代码存储库
- 构建过程的输出
- 构建过程报告

18.4.1 Jenkins 服务器配置

默认情况下，可以在 Jenkins 主服务器 IP 地址的 8080 端口找到 Jenkins UI。所以，如果 IP 地址为 192.168.10.10，在浏览器中输入如下 URL：http://192.168.10.10:8080。图 18-4 展示了第一次访问 Jenkins UI 时的初始屏幕。

我们需要做的第一件事是告诉 Jenkins 使用哪个 Java 开发工具包（JDK）进行构建。可以使用如下命令验证系统上的路径：

```
update-alternatives --list java
```

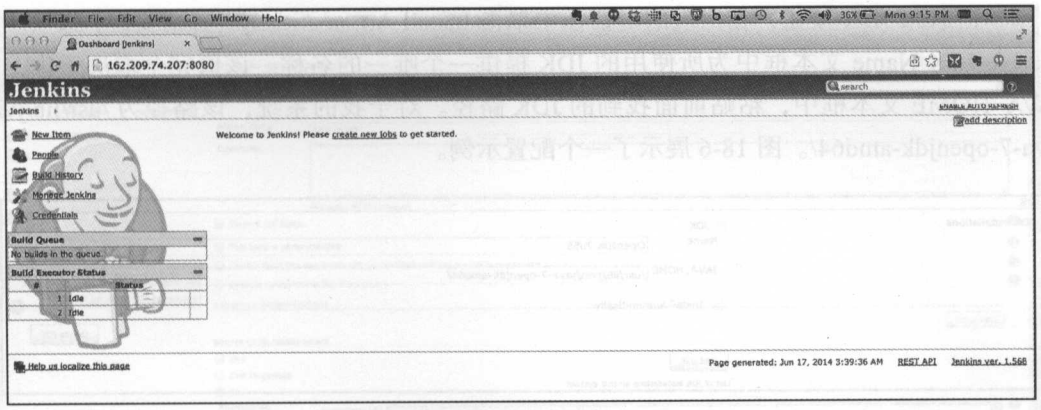


图 18-4　Jenkins 系统首页

在我的系统上，上述命令生成如下输出：

```
/usr/lib/jvm/java-6-openjdk-amd64/jre/bin/java
/usr/lib/jvm/java-7-openjdk-amd64/jre/bin/java
```

我使用 Java 7 进行测试，所以使用的路径是 /usr/lib/jvm/java-7-openjdk-amd64/。Jenkins 可以定义多个 JDK。但是，我们的测试应用程序不需要这一功能。

有了 JDK 路径，就可以告诉 Jenkins 这个位置。单击屏幕左侧的 Manage Jenkins 链接，然后单击下一屏幕中的 Configure System（配置系统）链接。图 18-5 展示了 Jenkins 系统的各种系统设置。我们只关心 JDK 设置。

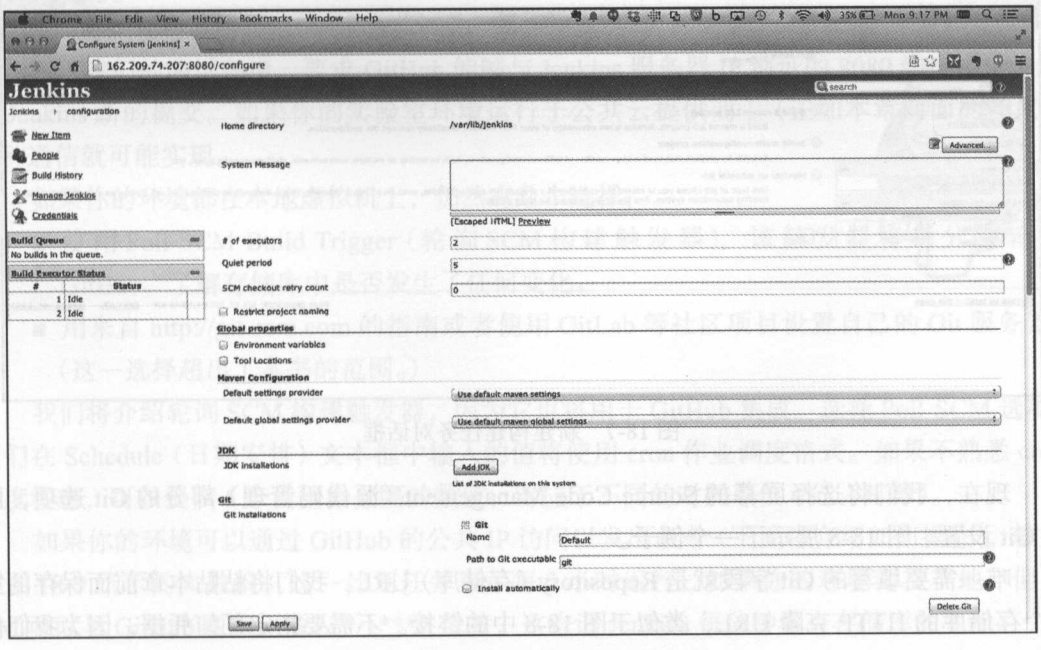


图 18-5　配置 Jenkins 使用的 JDK 路径

单击 Add JDK 按钮，在出现的对话框中单击 Install Automatically（自动安装）选项。接下来在 JDK Name 文本框中为所使用的 JDK 提供一个唯一的名称。该值完全取决于你，在 JAVA_HOME 文本框中，粘贴前面找到的 JDK 路径。对于我的系统，该路径为 /usr/lib/jvm/java-7-openjdk-amd64/。图 18-6 展示了一个配置示例。

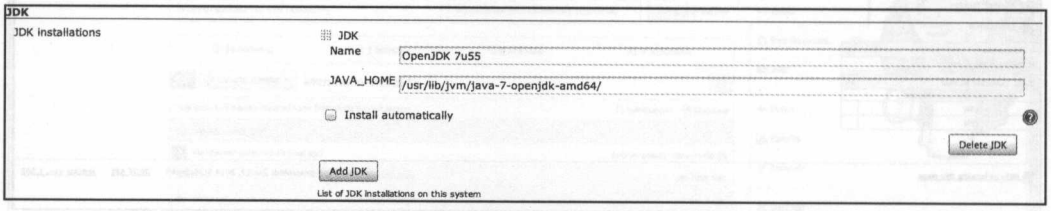


图 18-6 JDK 设置示例

单击 Save（保存）按钮，回到主屏幕。现在，我们已经为定义第一个构建任务做好了准备。

18.4.2 Jenkins 构建任务

单击屏幕左侧的 New Item（新建项目）。Jenkins 将询问构建项目的名称（Item Name 文本框）和想要创建的项目类型，如图 18-7 所示。我们将使用“free-style”（自由风格）选项以保持简单。任务的名称不需要与软件存储库名称匹配。单击 OK 转到余下的构建项目设置。

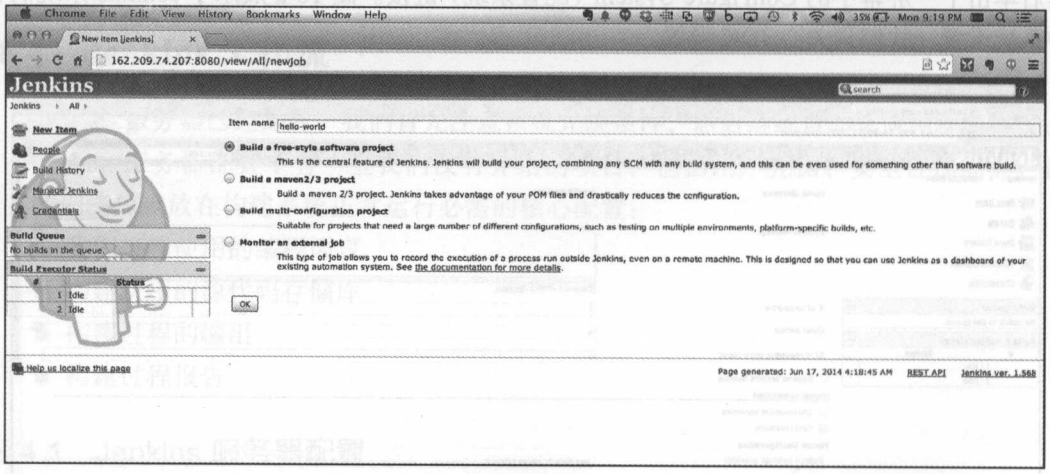


图 18-7 新建构建任务对话框

现在，我们将选择屏幕的 Source Code Management（源代码管理）部分的 Git 选项，配置 Git 设置。图 18-8 展示了一个例子。

唯一需要填写的 Git 字段就是 Repository（存储库）URL。我们将粘贴本章前面保存的第 4 个存储库的 HTTP 克隆 URL，类似于图 18-8 中的链接。不需要指定任何凭据，因为我们使用 HTTP 链接而非安全外壳（SSH）链接。

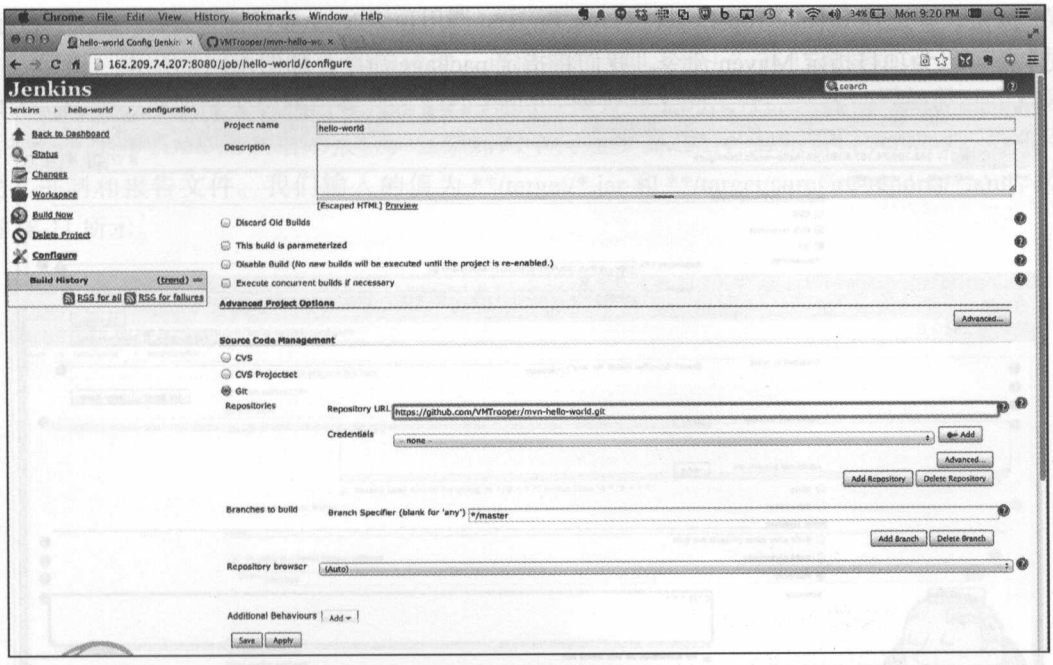


图 18-8　项目的 Git 存储库

图 18-9 中的 Build Triggers（构建触发器）部分确定构建任务的运行频率。理想情况下，应该配置 Git 服务器（我们使用 GitHub.com），在存储库更改时告诉 Jenkins，以便对每次存储库提交生成构建版本。这一功能通过 Git 的钩子功能实现，该功能可以在提交时触发操作。

对于我们的测试环境，要求 GitHub 能够与 Jenkins 服务器 IP 地址的 8080 端口通信，通知 Jenkins 新的提交。如果你的实验室环境运行于公共云提供商上（正如本章前面的建议），这种通信就可能实现。

如果你的环境都在本地虚拟机上，仍然有几个选择：

- 使用 Poll SCM Build Trigger（轮询 SCM 构建触发器），该触发器将每分钟轮询 GitHub，了解存储库中是否发生了任何变化。
- 用来自 http://git-scm.com 的指南或者使用 GitLab 等社区项目设置自己的 Git 服务器。（这一选择超出了本书的范围。）

我们将介绍轮询 SCM 构建触发器，因为它也将用于 GitHub 集成。选择 Poll SCM 选项，我们在 Schedule（日期安排）文本框中输入的值将使用 cron 作业调度格式。如果不熟悉 cron 调度程序，可以提供 5 个由空格分隔的数值，对应于不同的时间单位（分、时等）。

如果你的环境可以通过 GitHub 的公共 IP 访问以发出命令（下面将介绍如何操作），输入如图 18-9 所示的大数值。这一计划任务表示每 3 小时运行。如果你的环境都在本地，但是 VM 可以与 GitHub 通信，输入 */1**** 代替，这告诉 Jenkins 每分钟检查存储库的变化。

接下来是构建的步骤。在页面的 Build（构建）部分，单击 Add Build Step（添加构建步

骤）按钮，并选择 Invoke Top-Level Maven Targets（调用顶级 Maven 目标）选项。Goals（目标）文本框可以为项目指定 Maven 命令。我们将指定 package 命令，如图 18-10 所示。

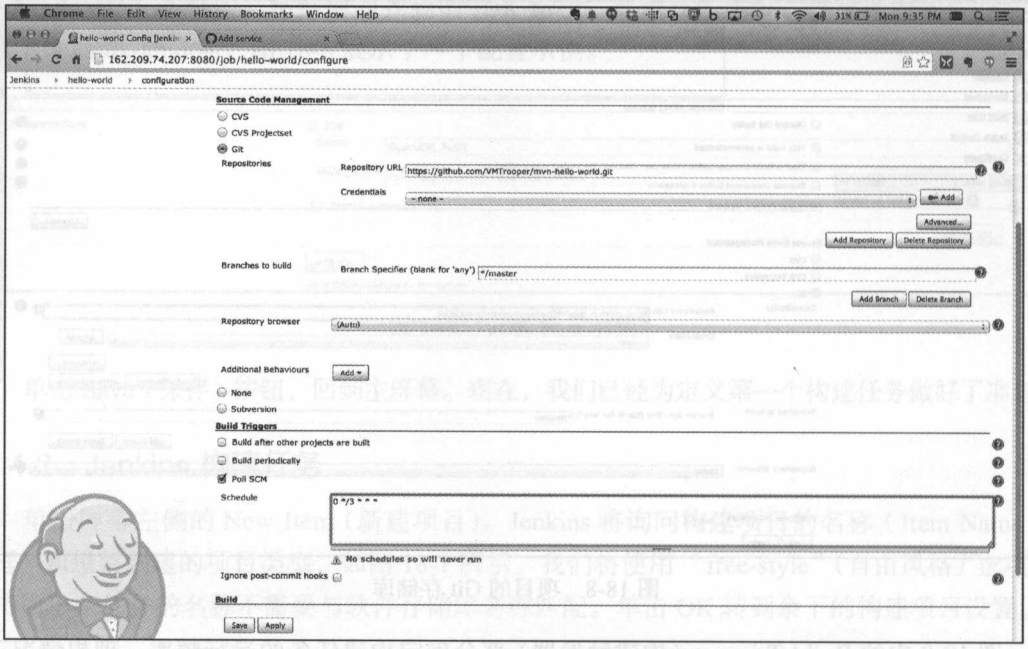


图 18-9 Jenkins 存储库构建触发器

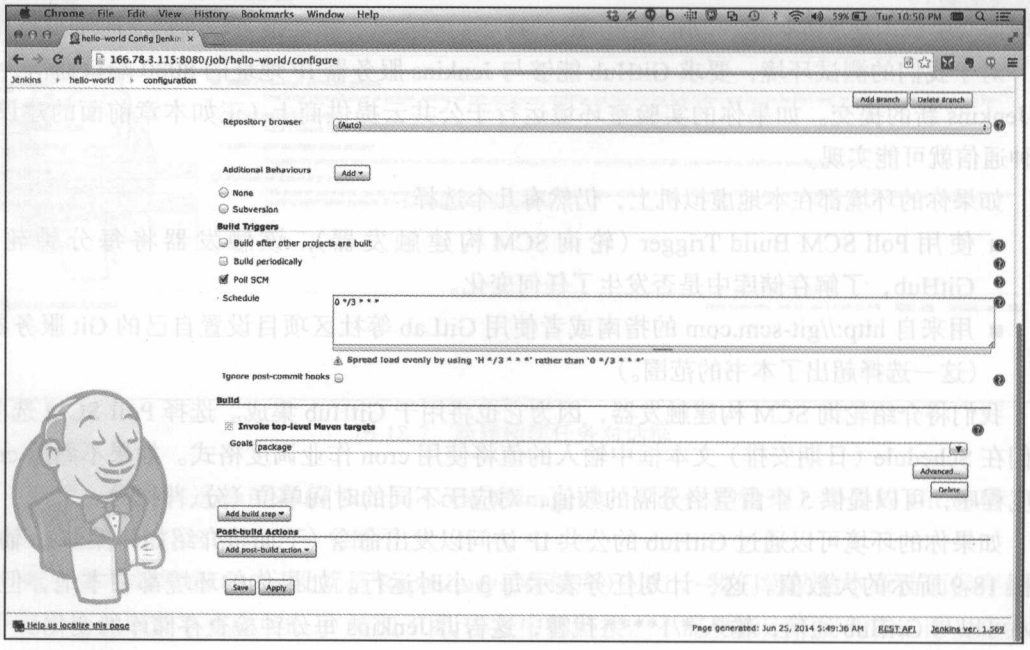


图 18-10 Jenkins 构建步骤

如果想要运行多个构建或者测试步骤，可以定义任意数量。

现在，我们如何获得报告和编译的二进制程序（即工件）？我们需要定义一些构建后操作。首先，添加一个构建后操作，归档 Jenkins 服务器上的工件。其次，添加另一个构建后操作，发布 JUnit 测试结果报告。在添加两个构建步骤之后，必须告诉 Jenkins 在哪里寻找二进制和报告文件。我们输入的值为 **/target/*.jar 和 **/target/surefire-reports/*.xml，如图 18-11 所示。

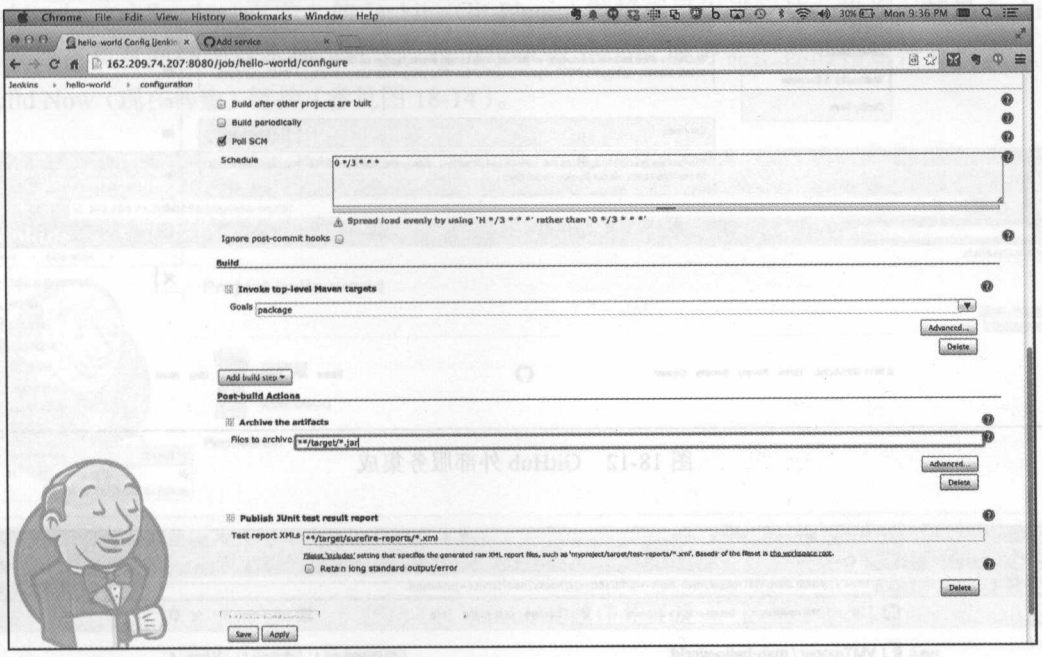


图 18-11　Jenkins 构建后操作

在我们指定的路径中，** 表示“你好，世界”Maven 项目的根目录。目标目录在第一次执行 Maven 打包命令（如果在命令行上运行，是 mvn package）之后创建。编译后的程序将保存在目标目录下的一个 jar 文件中，来自 Jenkins 构建的 JUnit 测试报告将放在目标中的 surefire-reports 子目录。

第一个 Jenkins 构建任务已经做好准备保存和执行，单击 Save（保存）。

18.4.3　Git 钩子

如果 VM 有可公开访问的 IP 供 GitHub 通信，可以在 GitHub.com 上配置一个提交后 Git 钩子。转到 mvn-hello-world 知识库中的分支，并单击 settings。现在，单击页面左侧的 Webhooks & Services（Web 钩子和服务）链接，可以看到图 18-12 中显示的页面。

单击 Add Service（添加服务）按钮并选择 Jenkins(Git Plugin)。不要选择 Jenkins(GitHub Plugin) 选项，因为我们在 Jenkins 服务器上安装了通用的 Git 插件。我们选择通用 Git 插件代

替 GitHub 版本，以防需要使用其他 Git 服务（如 gitorious、BitBucket 或本地托管的 GitLab 实例）的情况（参见图 8-13）。

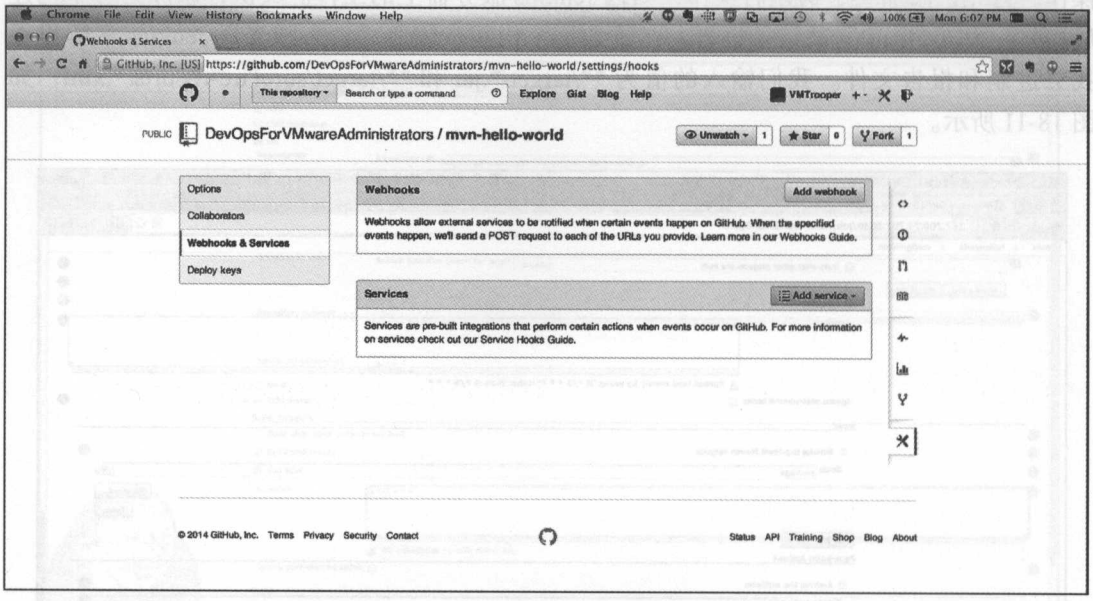


图 18-12 GitHub 外部服务集成

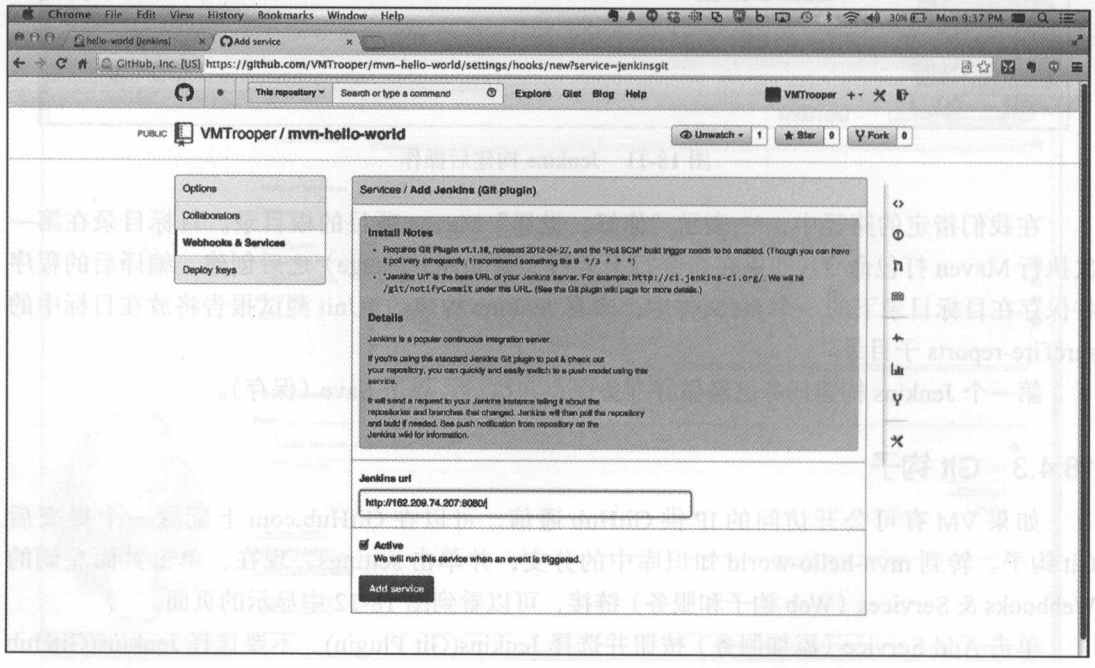


图 18-13 Git 提交后钩子设置

配置相当简单，只需要提供 Jenkins 服务器的 URL，确保选择 Active 选项，并单击 Add Service 按钮。

18.4.4　你的第一次构建

现在我们已经配置了构建任务，可以生成第一个工件了。如果 GitHub 可以访问你的 Jenkins 服务器，就可以测试这一服务，真正地从那里触发构件。单击存储库设置中的 Webhooks and Services 页面上的 Jenkins(Git Plugin) 链接就可以进行这一操作。如果 Jenkins 服务器在私有网络中的本地 VM 上，也可以单击 Jenkins 服务器上你的构建任务网页上的 Build Now（现在构建）链接（参见图 18-14）。

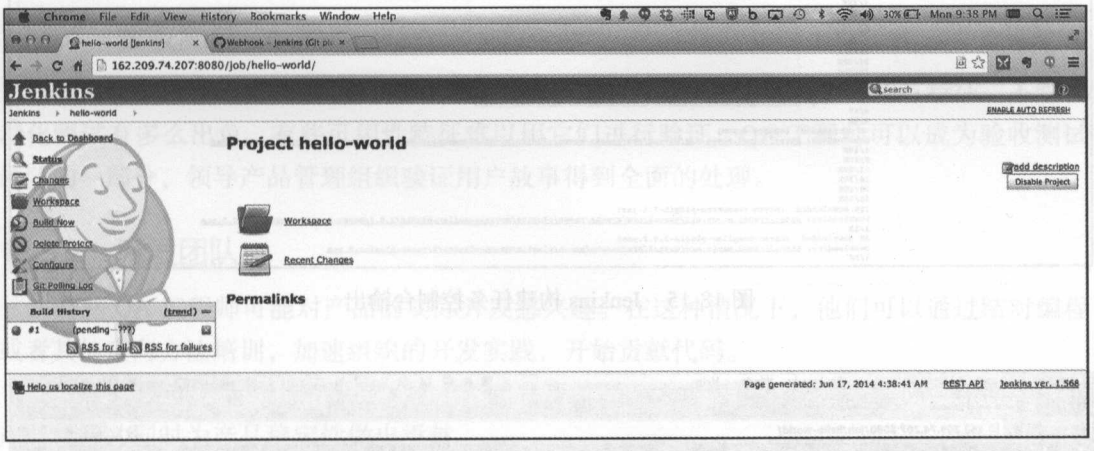


图 18-14　Jenkins 构建任务页面

不管使用哪种方法触发构建，Jenkins 将立刻开始编译源代码并生成报告。当屏幕左侧的 Build History（构建历史）窗口中的任务变成活动状态时，构建进度变成一个可点击链接，将你带入当前任务执行。单击该链接，然后单击后续页面上的 Console Output（控制台输出）链接，可以检查构建进度。图 18-15 展示了一些输出示例。

注意，构建执行的第一行说明了构建运行的原因，Jenkins 首先克隆源代码存储库，然后运行 mvn package 命令。

构建成功时单击 Back to Project（返回项目），构建项目的首页上应该至少有两个附加链接：一个用于 jar 文件工件，另一个用于测试结果（参见图 18-16）。

如果在桌面上本地安装了 Java 7，可以真正下载 jar 文件，用如下命令执行：

```
java -cp target/mvn-hello-world-1.0-SNAPSHOT.jar com.devops.app.App
```

如果没有对源代码进行任何更改，输出应该如下：

```
Hello DevOps World!
java -cp target/mvn-hello-world-1.0-SNAPSHOT.jar com.devops.app.App
```

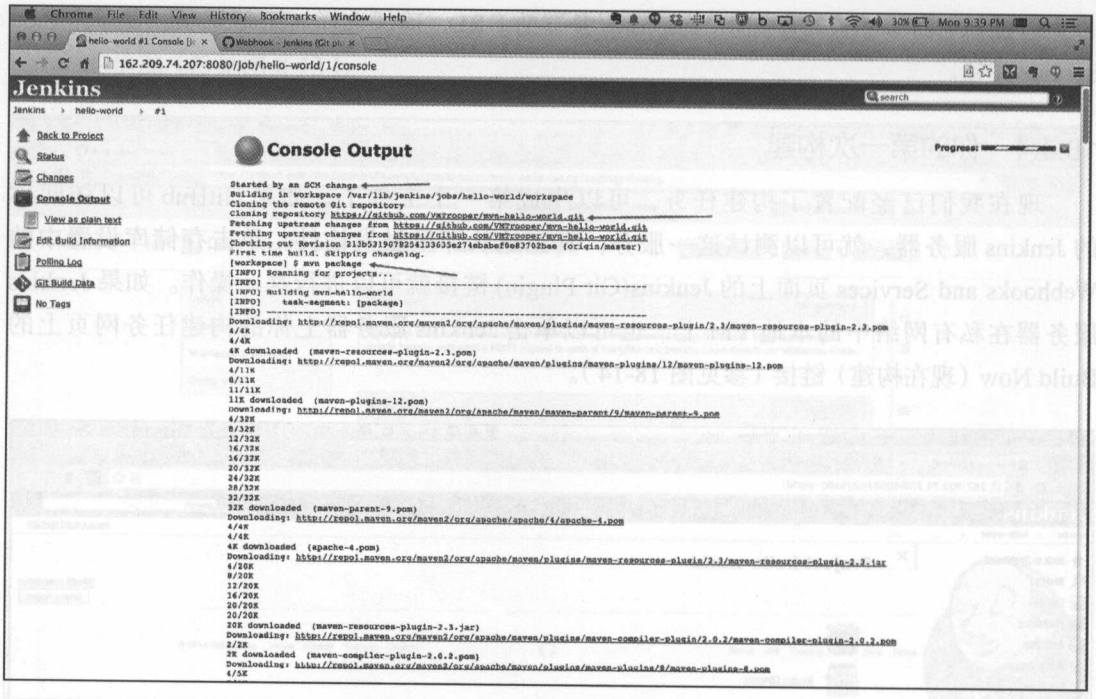


图 18-15 Jenkins 构建任务控制台输出

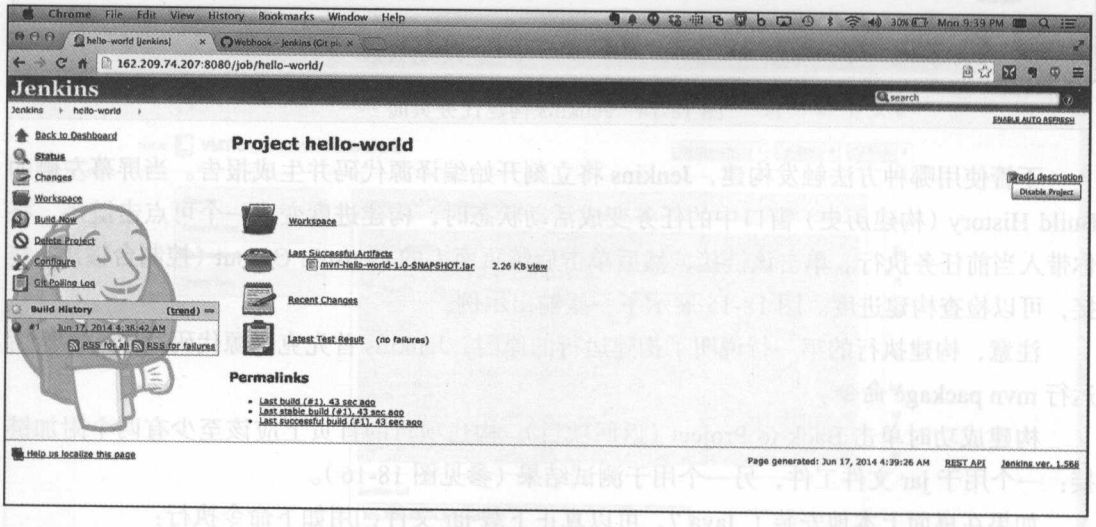


图 18-16 任务执行之后的构建工件和报告

试验源代码（mvn-hello-world/target/src/main/java/com/devops/app/App.java）并将更改推送回 Git 服务器（GitHub 等）上的存储库。如果 Git 钩子有效，提交到 GitHub 存储库会立即触发 Jenkins 服务器上的新构建任务。否则，在下一分钟，Jenkins 服务器将注意到 GitHub 存

储库的更改，并自动触发构建。

此时，你的构建工件已经为生产环境中的预演做好了准备。所以，可以添加另一个构建后操作，让 Maven 创建一个正式发行版本，并将该工件放置在存储库服务器。如果你的小组正在实践 CI，构建将经过最后一轮验证并投产。如果你的小组正在实践 CD，工件将直接投产。

18.5　质量保证团队

传统软件开发组织有专门的质量保证（QA）团队。CI 系统可以避开 QA 人才的需求吗？绝对不是！我们的 QA 工程师可能成为组织中其他团队的重要成员。

18.5.1　验收测试

产品管理团队通常负责验证软件发行版本符合他们为开发团队布置的所有特征。不管自动化测试有多么出色，有些可用性特征难以用它们进行验证。QA 工程师可以成为验收测试团队的一部分，领导产品管理组织验证用户故事得到全面的处理。

18.5.2　开发团队

有些 QA 工程师可能对产品的实际开发感兴趣。在这种情况下，他们可以通过结对编程或者其他定向方法培训，加速组织的开发实践，开始贡献代码。

缺陷修复可能是一个很好的出发点，这些新的开发人员可以了解编码惯例，在彻底研究产品架构的同时为产品稳定性做出贡献。

18.5.3　构建 / 测试基础设施

如果 QA 工程师享受软件开发过程，但是更关注测试过程，运营团队总是可以从他们的软件开发专业技能中获益，维护持续集成基础设施。

18.6　小结

如果手工进行软件包装和部署，可能很麻烦且容易出错。Jenkins 等持续集成 / 交付系统可以自动化代码下载、编译、测试和打包过程，消除人为错误。

参考文献

[1] Apache Maven documentation: http://maven.apache.org/

[2] Jenkins CI documentation: http://jenkins-ci.org/

第八部分 Part 8

VMware DevOps 实践

- 第 19 章　DevOps 环境中的 VMware vRealize Automation

Chapter 19 第 19 章

DevOps 环境中的 VMware vRealize Automation

到了本书的这个阶段，对 DevOps 的定义已经进行了很多交流。VMware 在这个主题上的愿景是，对此没有客观的“真正”定义，而是倡导将 DevOps 看作迈向开发和运营团队之间通力协作状态的一段旅程。虽然没有具体的“工具”能够定义它，但是 DevOps 以及真正的软件定义数据中心的核心组成部分之一是扩增现有工具集功能的可扩展平台。

本章包含如下主题：

- DevOps 的出现
- 稳定的敏捷性
- 人、过程和 Conway 法则
- vRealize Automation
- vRealize Application Services
- Puppet 集成

19.1 DevOps 的出现

根据从客户那里得到的经验，我们发现了云走向成熟的一个模式。这一段旅程通常有几个阶段，但是不总是顺序出现的：

1. 自动化虚拟基础设施。
2. 启用门户以提供服务目录；该门户集成到基础设施即服务（IaaS）组件的消费引擎中。

3. 提供应用程序蓝图服务，允许简单应用程序、中间件、数据库和复杂多层应用程序配给标准化，增进应用程序团队的价值。

4. 提供高级、成熟的功能，如应用程序发行自动化。

5. 最后，将 IT 部门确立为真正的服务代理。

DevOps 作为一个学科出现，本身就是行业中令人兴奋的一件事情。我们看到了将业务和 IT 运营合理化提升到前所未有高度的机会，在开发人员工具领域中没有清晰的“成功技术”。VMware 热心地观察局势的发展，积极致力于和许多成熟和新兴的工具（如 Puppet、Chef、Artifactory、Jenkins、Docker 等）集成。每天都有新的工具流行起来，但是 VMware 坚持保持框架的开放性，在更广泛的部署和代理服务环境中有效、巧妙地利用这些工具。

19.2　稳定的敏捷性

敏捷性是一个“性感”的概念，刚性则不然。IT 行业的发展速度令人难以想象，各家公司都在拼命寻找完美的解决方案以满足其目标的要求，而它们的目标又往往处于变化之中。而且，企业往往需要做出影响之后 3 年（甚至更久）的购买决策。领导人没有水晶球，所以通常（也是正确的）寻求能够适应业务需求的灵活和可扩展解决方案。

但是，这只是事实的一个方面。少量的刚性——或者更大方一点说，长期的稳定性——对于负责任的运营工作来说是至关重要的。正如航空母舰上 F-18 战斗机起飞时那样，支持数据中心的稳定基础设施是任务成功的关键。有些更改必须很谨慎地进行，往往需要几个月才能很好地规划。数据中心存在更改控制等概念有很充足的理由，许多错误无法立刻回滚，而运行中断直接影响业务，这种影响往往是灾难性的。

相反，当今许多应用程序的变化非常快。对于 Netflix 这样的公司，每天都要实施和投产许多新功能、更新和缺陷修复，无缝地传递给最终用户。持续集成（CI）和持续交付（CD）等方法越来越令人满意，并且通过对不成功代码能够在必要时立即回滚的理解，支持“快速失败，经常失败”的哲学。这种立刻转向的能力是敏捷型企业所寻求的，但是需要花费精力创建支持这一能力的底层结构。

按照前面的类比，如果有选择的话，海军可能很愿意采用更快、更敏捷的航空母舰。确实，如果底层基础设施是航空母舰，虚拟化和软件定义数据中心实现的就是这种发展。但是，航空母舰的运行仍然需要谨慎为之，并进行足够的规划。不管如何升级，即使最好的航空母舰在机动性上也明显不可能胜过 F-18；它们的设计目的不同。

19.3　人、过程和 Conway 法则

19.2 节暗示，今天的许多组织中出现了某些冲突。我常常看到开发和运营团队分坐在桌

子的两边，从物理上、逻辑上和概念上分处不同区域。有时候紧抱双臂，有时相互指责。这应该是寻求凝聚力的组织所担心的事情。

任何组织在设计系统时，设计的结构都会复制组织的沟通结构。

应用程序架构师和开发人员希望快速行动，通常不特别关心开发所用的框架基础；他们只想展翅翱翔。因此，平台即服务（PaaS）产品相当受欢迎。基础架构和运营团队往往因为前面所提到的约束而无法适应开发人员不断变化的需求——也就是说，确保控制措施的持续，为企业获得更大利益。因此，运营团队被认为动作迟缓，是不受欢迎的“影子 IT”现象的成因；如果开发人员需要快速完成某项工作，他可能使整个组织控制知识产权的努力“短路”。很明显，这种情况必须得到解决。

IT 专业人士总是悄悄地围绕那些使工作变得更轻松的人，自发地组织起来，同时避开那些使工作变得更加艰辛的人，这与组织结构图无关。

在 VMware，我们支持这样一种理论：当对人和过程的改变得到以坚持类似原则为目标的技术的加强时，这些更改最有效。软件定义数据中心（SDDC）就是这样一种机制：我们通过完全以更容易改编的软件构架基础设施解决方案，尽可能自动化过程，同时保持安全和负责任，大大提高了“航空母舰”的速度。

在这里我们还要关注另一种机制，这是对 SDDC 愿景的补充，旨在用公共的框架将应用程序和基础架构团队组合在一起，他们可以利用这一框架提取价值、共享任务、紧密协作，以实现业务目标。这种技术上的相近性培养了团队精神，有助于更加顺畅、快速地进入市场。下面我们深入观察一些此类方法和工具集。

19.4 vRealize Automation

IT 消费者在过去几年已经有了显著的发展，这在很大程度上归功于在 Apple 激发下兴起的 App Store 概念。各类消费者期望可以按需获得服务，最大限度地减少麻烦，这就要求以一种自动化和治理措施作为服务目录的基础，及时满足需求，且具备合适的安全保护。

vRealize Automation（vRA）原称 vCloud Automation Center，是更广泛的 vCloud 套件的一个重要组件，它是自动化和治理活动的中心，同时提供用户友好的服务目录。消费者请求通过一个门户，在此后与其工作负载交互，如图 19-1 所示。

vRA 的基本价值主张体现在解决方案的可扩展特性。衰退期（fading）是单独模板的部署足以符合大部分业务需求所需的天数。模板部署本身是服务提供的一小部分；IP 地址管理（IPAM）自动化、防火墙和负载平衡器集成、备份策略分配、根据策略将集成多层应用程序部署到异构的内部和外部基础设施——这一切都正在接近真正的现代 IT 代理。

任何人如果曾经犯过错误，让自动化过程在没有经过合适的检查和平衡之前就投入运行，就会了解有策略的过程人为监控的价值。完全由管理员定制的健全、细致的批准引擎，能够确保影响较小的请求立即得以实施，而资源密集或者特权请求被加上标记，在机器生命期的任何阶段人工批准。

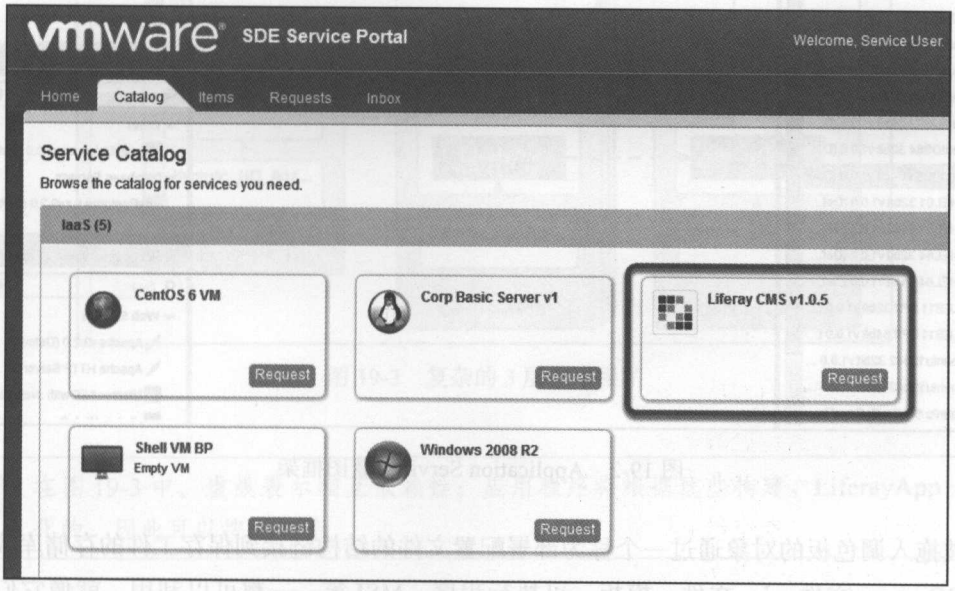


图 19-1　vRealize Automation 服务目录

但是我们在大多数情况下仍然会谈到基础设施的渠道问题，要如何消除基础设施团队和开发团队之间的隔阂？

19.5　vRealize Application Services

vCloud 套件中与 vRA 紧密集成的组件是 Application Services（以前称为 Application Director 或 AppD）。Application Services 提供一个框架，在此框架内可以使用直观的拖放调色板初始化组件及关系，构建全面集成的多层应用程序栈。

应该将这一层次视为部署编排引擎，是创建和部署应用程序蓝图的一项标准化、灵活、与底层基础架构解耦的机制。图 19-2 展示了蓝图的一个示例。


注意　Application Services 蓝图框架由如下部分组成：1. 包含一个 CentOS 模板的应用程序蓝图工作空间。2. 这个模板的配置细节。3. 此列包含应用程序组件、外部服务和原生服务的模板。4. 此列包含连接到云提供商的逻辑模板。

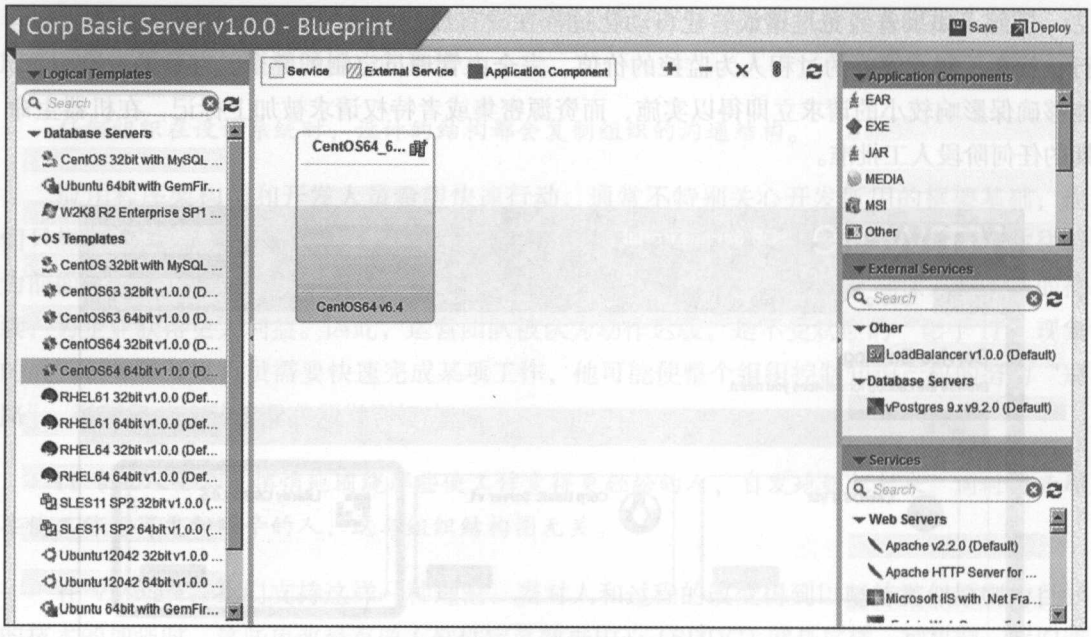


图 19-2 Application Services 蓝图框架

被拖入调色板的对象通过一个称为部署配置文件的结构链接到保存工件的存储库或者共享。ISO、war 文件、jar 文件、模板、可执行程序、MSI 等——都可以利用，就像它们是应用程序构建过程的原生组件那样。

最终，构建必须在基础设施上“着陆”，因此部署配置文件与某个云提供商关联，允许将相同的蓝图应用到不同环境的多次部署上。这些环境可以是 vRealize Automation、vCloud Director 或 Amazon EC2，为了简单起见，我们将专注于 vRA 集成，但是了解应用程序蓝图的可能性和固有可移植性很重要。

那么，这对部署有何影响？很简单，每个以这种风格塑造、从服务目录中部署的服务都是新鲜、活跃、按需制作的。这为应用程序架构师提供了灵活性，可以创建和修改具备复杂相互依赖性的成熟集成应用程序栈，自动化构建过程，并将这些作为目录项输出给消费者，显著缩短实现价值和请求实施的时间，消除许多容易造成错误的重复任务；构建的标准化和自动化确保了最终产品的稳定性。

这一解决方案集中值得注意的特性之一是使用 Application Services 向内外伸缩的能力。如图 19-3 所示，应用程序架构师可以将一个节点标记为可伸缩，实际上可以通过用户界面（UI）或者具备所需负载平衡能力的应用程序编程接口（API）按需伸缩该节点。这种功能可以供 vRealize Operations（以前称为 vCenter Operations Manager）等性能监视技术利用，根据工作负载或者其他关键性能指标（KPI）条件实现伸缩。

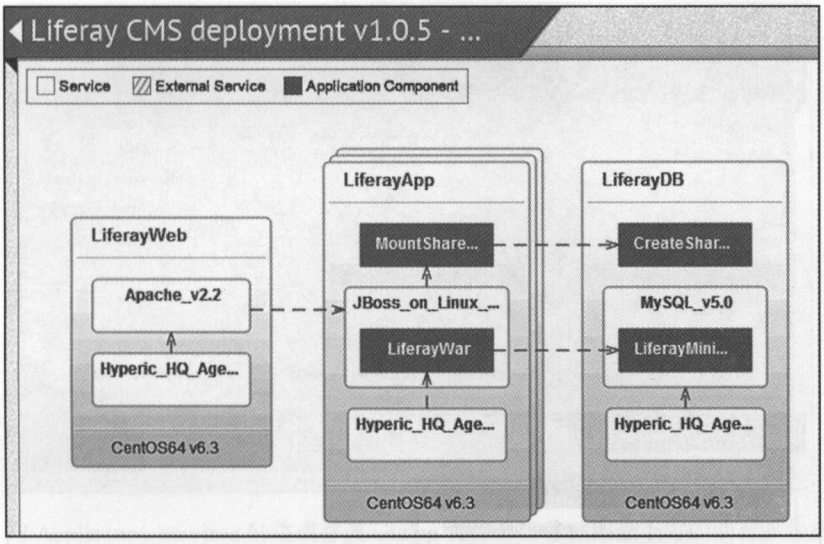


图 19-3　复杂的 3 层应用程序

注意　在图 19-3 中，虚线表示相互依赖性；应用程序将根据这些构建。LiferayApp 是群集化的，因此可以按需伸缩。

回到 Application Services 中的调色板概念，考虑下一步如何构建应用程序。记住，可扩展性是 VMware 的关键原则！我们通过与 Puppet 的集成进一步研究这一原则。

19.6　Puppet 集成

VMware 是 Puppet Labs 的早期投资者，与他们有着密切的伙伴关系，结果促成了一个补充框架，对两种技术的消费者都有好处。

注意　Puppet 有一个企业版本，但是与 VMware 解决方案的集成并没有明确要求这一版本。不过，你应该始终与 Puppet Labs 确认 Puppet 许可证的相容性。

Puppet 利用主机 / 代理层次架构，便于大规模应用和通过单一界面进行 OS 控制。当代理上线，它们可以自动加入 Puppet 主机，此时它们将接受到指定的清单。根据清单中的条件，代理可以开始进行系统更改，并从主机拉取模块，与预期状态达到相容。

Application Services 允许 Puppet 以解决方案的形式注册（参见图 19-4）。这种集成使 Puppet 模块得以在 Application Services 内部通过拖放调色板作为原生对象利用。

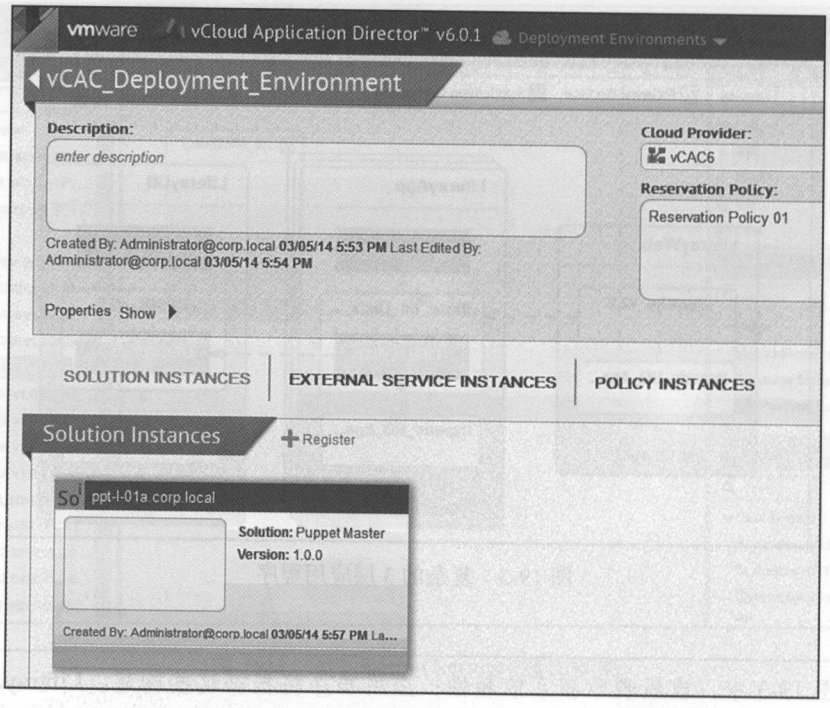


图 19-4 在 Application Services 中注册的 Puppet 解决方案

服务的编排和部署由 VM 本身和全面集成的应用程序栈组成，现在可以自动注册到 Puppet 主机，开发团队可以管理这些实例的更新，并监控配置的漂移，在必要时实施补救措施。

这种集成的净效应之一是基础设施和开发团队之间的耦合更为紧密——这是 DevOps 的核心原则之一。两个团队一起工作是有利的，因为该框架对两个团队都有好处。

那么，这是怎么发生的？要将 Puppet 注册为 Application Services 中的一个解决方案，可以查看 VMware KB 2068342，它提供了与解决方案当前版本相关的最新信息。最后，必须登录到 Puppet 主机并下载和运行 KB 中提供的 Ruby 脚本。下面是注册 Application Services 6.1 的例子。

```
ruby RegisterWithAppD.rb -i Application_Director_IP -u User_Name -p
Password -t Tenant_Name -g Group_Name -d Deployment_Environment_Name
```

让我们来了解一下上述命令的运作过程。首先，从 Puppet 主机运行一个简单的 Ruby 脚本，将 Puppet 加入 Application Services，作为一个解决方案。然后，你会看到 Puppet 注册为解决方案，如图 19-4 所示。

在 Puppet 端发生的是，这一集成修改了 Puppet 中的 site.pp 文件，该文件是主配置文件，修改的目的是加入与这台 Puppet 主机集成的所有 Application Services 节点目录（参见图 19-5）。

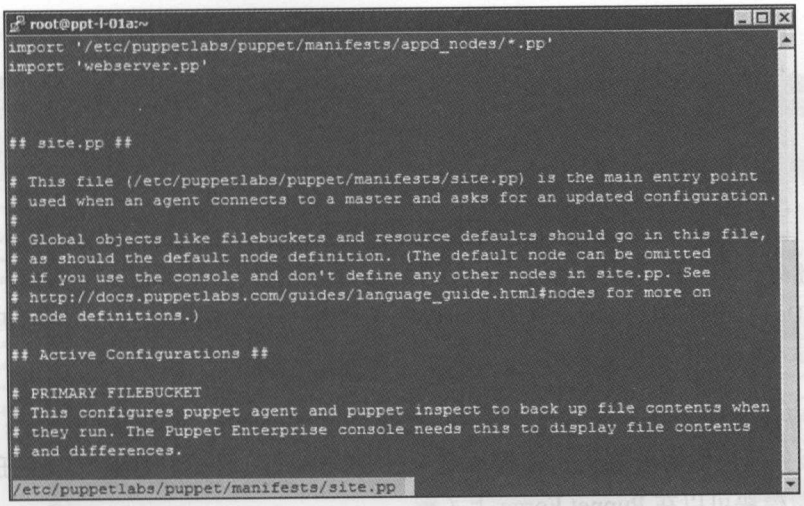


图 19-5　与 Application Services 的集成修改 site.pp 文件。屏幕截图最上方的两行是集成前加入的

接下来，我们需要将 Puppet 模块导入 Application Services（参见图 19-6）。

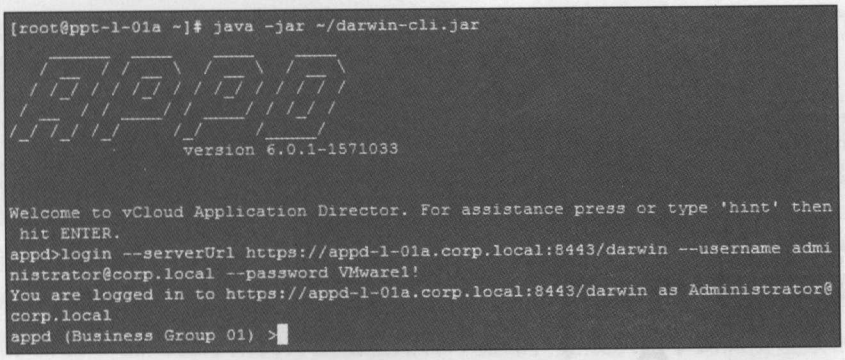


图 19-6　从 Puppet 主机启动 Darwin 客户端

执行如下步骤，将 Application Director 注册到 Puppet：

1. 按照图 19-5，从 Puppet 主机启动 roo shell 客户端，登录到 Application Director 远程 shell：

```
java -jar ~/darwin-cli.jar
```

2. 以管理权限登录到 Application Services 框架：

```
Login --serverUrl https://FQDN:8443/darwin --username Admin@domain
--password PASSWORD
```

3. 按照图 19-7，导入所要使用的 Puppet 模块：

```
import-puppet-manifests --puppetPath /usr/local/bin/puppet --typeFil-
ter "^motd$"
```

```
appd (Business Group 01) >import-puppet-manifests --puppetPath /usr/local/bin/pu
ppet --typeFilter "^motd$"
Puppet Resource                                                          Ser
vice Name                                                              Import
 Status
motd                                                                     Pup
pet motd                                                               IMPORT
ED

Number of new service versions created: 1
Number of already existing service versions: 0
appd (Business Group 01) >
```

图 19-7 上述命令可以导入任何符合正则表达式“^motd$”的模块，在此该表达式表示“Message of the Day”（当日消息）Puppet 模块。可以使用通配符导入所有模块，但是这样做应该小心，因为导入文件的尺寸可能很大

这时，应该看到 Puppet 模块已经导入 Application Services，如图 19-8 所示；Application Services 的服务门户中列出的 Puppet 模块是 MySQL Server。但是，这也可以是任何模块。社区构建的免费模块可以在 Puppet Forge 上了解。

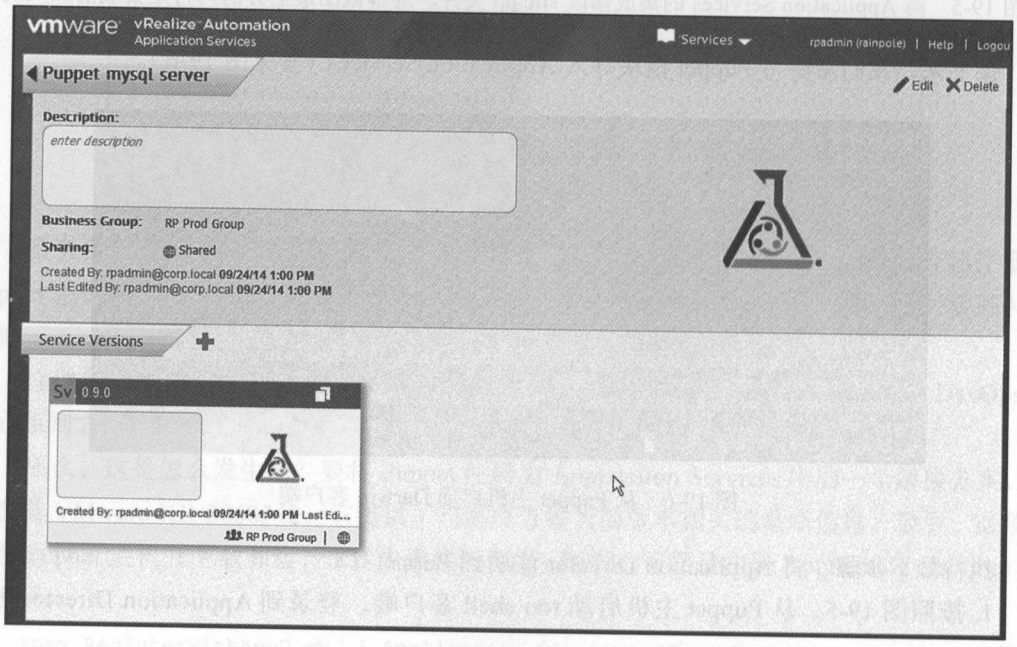


图 19-8 现在，可以将 Puppet 模块作为原生服务利用

根据模块的性质，你可能需要从 Application Services 中为模块定义添加一些信息。插入点允许从 VMware 或者其他解决方案传递变量给 Puppet 模块，自定义和自动化应用程序部署。同样，需要两个团队之间的协调，以确保必要的变量从合适的位置拉取，交付给应用程序。

在这一阶段，Puppet 模块可以在 Application Services 中原生使用。如图 19-9 所示，MySQL 服务以对象的形式出现，可以添加到 CentOS 实例或者任何其他兼容实例，添加的次数根据应用程序蓝图的要求而定。记住，因为这些逻辑模块都是解耦的，它们有很好的移植

性和灵活性。

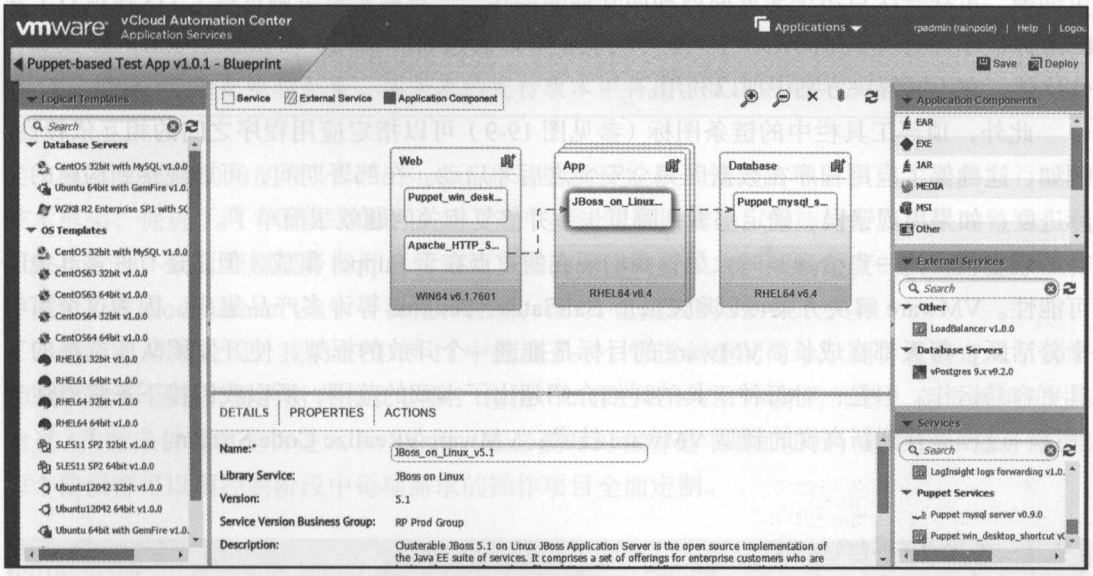


图 19-9　Puppet 服务器现在以元素的形式出现，可以灵活地部署在应用程序蓝图中。左上角可以看到可用的 Puppet 服务

相反，如果你只想在预装 Puppet 代理的情况下利用模板，可以简单地在任何阶段将代理部署到模板化的虚拟机（VM）上。前面所概述方法的真正价值是创建跨越多个 VM 的可移植、多层、完全集成的应用程序，同时仍然易于可视化和消费。在初始设置之后，应用程序构建可以有效地转交给对 Puppet 一无所知的团队，使 Puppet 管理员无需参与构建过程。

这种解决方案的潜力应该是显而易见的。基础设施团队简化了主蓝图的创建和部署，并向消费者提供服务。应用程序架构师控制应用程序蓝图，完成从简单到高度复杂的应用程序栈的所有设计。一旦投入运营，应用程序可以通过 Puppet 进行管理、更新、补丁以及配置漂移控制。

图 19-10 中可以再看到一个基于 Liferay 应用程序的部署后多层蓝图。

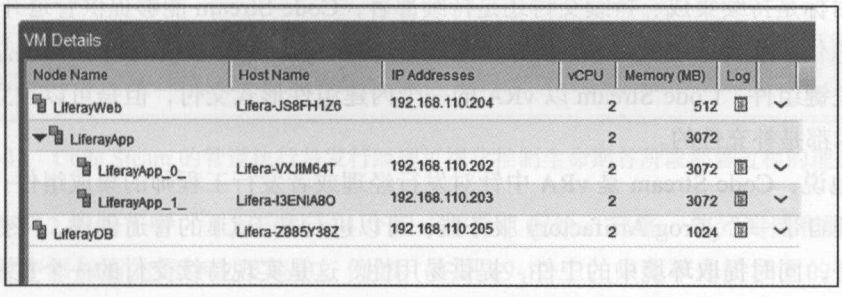


图 19-10　在 Application Services UI 中可视化部署后的 3 层可伸缩应用程序栈

在这里，可伸缩节点（LiferayApp）的加入很重要。Application Services 将标记该节点为可伸缩，可在一次点击中完成向内和向外伸缩操作——只需要单击调色板上 OS 模板右上角的服务器小图标（参见图 19-9）。这一功能也可以通过 API 访问，可以联系 VMware 及其合作伙伴，在不同解决方案中加以利用。

此外，顶部工具栏中的链条图标（参见图 19-9）可以指定应用程序之间的相互依赖性。例如，这确保了应用程序在数据库完全安装之后才启动。在部署期间，可以观察到构建的实际进度；如果出现错误，确定部署在哪里失败并修复相关问题就很简单了。

现在提出这一点恰逢其时：尽管我们现在的重点在于 Puppet 集成，但是还有许多其他的可能性。VMware 解决方案可以和 Chef、SaltSatck、Jenkins 等许多产品集成，因为这个市场十分活跃，每天都在成长。VMware 的目标是推进一个开放的框架，使开发团队所选择的工具更容易运作。但是，对每种工具的详细介绍超出了本章的范围，所以我们接下来将专注于使这一过程提升到新高度的新型 VMware 技术：VMware vRealize Code Stream。

19.7 Code Stream

详细介绍即将出现的 Code Stream 解决方案之前，我们先定义一些术语：

- 持续集成指代码在存储库中的定期登记，以及为确保及早发现缺陷，以最小代价加以修复而进行的构建及测试自动化。
- 持续交付建立于强大的持续集成基础之上，是一系列确保最新代码可在任何时候推送到生产环境的实践，但是推送到生产环境的过程仍然可能是人工步骤，而且不一定推送所有代码更改。自动化构建进行全面的测试；持续交付过程中代码会通过从开发到测试，再到用户验收测试（UAT）的门控机制。
- 持续部署是每次登记都被自动推送到生产环境的状态。这一模式的基础方法和持续交付大致相同，但是发挥到极致。这一模式明确地要求对自动化构建和测试过程的最大信心，也要求企业能看到超出该模式潜在风险的好处。Etsy 和 Facebook 可能将此看作最终目标，但是银行可能不将其作为追逐的目标。最终，一切都取决于业务需求。

不管目标是持续集成、持续交付还是持续部署，Code Stream 能够提供管道开发自动化、利用现有源代码管理系统（如 Git）、构建系统（如 Jenkins）和工件存储库，提供总体解决方案的一个关键组件。Code Stream 以 vRA 的一个内建组件形式交付，但是可以独立运营，不过这些技术都是补充性的。

确切地说，Code Stream 是 vRA 中针对发行经理或者发行工程师的集成组件；除了核心代码，它还自带一个 jFrog Artifactory 服务器，可以进行基于 UI 的管道创建、门控和特定环境中的部署，同时提取环境中的工件，提供易用性。这是实现持续交付的一个重要步骤；自动化管道控制促进了开发阶段之间的无缝过渡，确保坚持最佳实践和标准，同时利用现成或

者脚本化工具进行 CI 测试。

> **注意**　在 Code Stream 的 1.0 版本中，只支持自定义脚本、vRA、vCenter Server 和 Cloud Director 的配给和配置。在未来的发行版本中将提供更多的解决方案和集成点。

从人和过程的角度看，Code Stream 通过提供开发和运营团队可以参与实现共同目标的技术框架，促进了两个团队之间的互动。这个共同的目标就是：保持与开发团队的整合，包括发行管理，同时理解服务器和运营方面，是确保不同阶段之间顺畅过渡的关键。这个工作空间可以在 Code Stream UI 中可视化和管理，该 UI 也集成到 vRA UI 中。

为了直观地说明，我们在图 19-11 中提出了一个管道创建的简单示例。管道定义了部署和测试应用程序所需的所有必要步骤，包括（如有必要）VM 本身的部署。在步骤推进中，通过 / 失败条件确定是否进入下一阶段。在必要时可在管道中增加阶段，这取决于业务需求。每个阶段都可以用控制阶段中每项需求的操作项目全面定制。

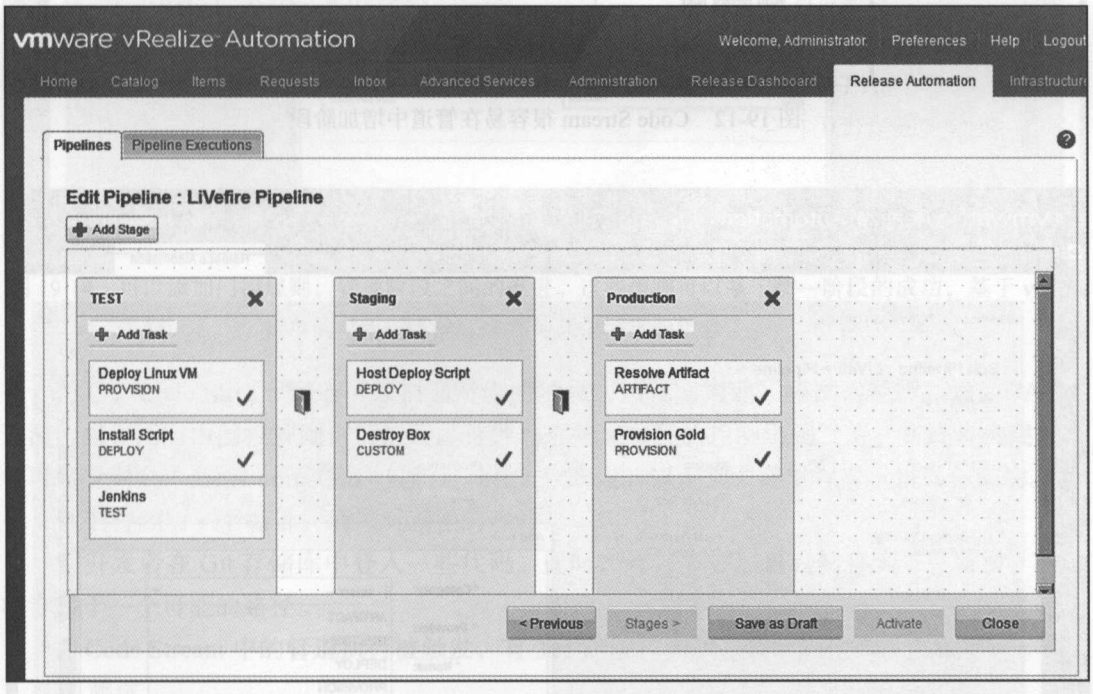


图 19-11　Code Steam 的管道执行是发行经理可视化控制生命期各阶段部署过程的理想框架

在图 19-12 的例子中，开发阶段有 4 个组成部分：配给、部署、单元测试和脚本测试。

在阶段中添加新的操作也很简单。例如，在图 19-12 中，只要单击阶段上的 + 符号并从下拉式菜单中选择（参见图 19-13）。这种灵活性很重要，例如，后续的阶段可能不需要完整

配给的服务器；它们可能只需要在现有操作系统基础上的增量构建。

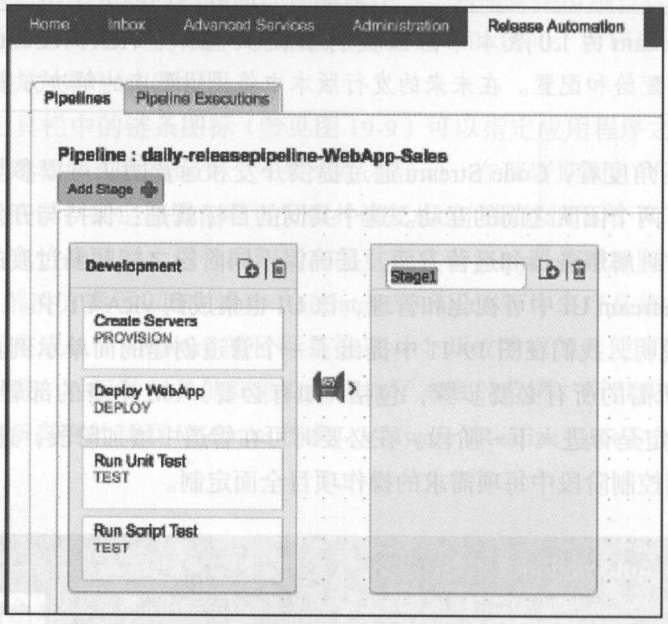


图 19-12 Code Stream 很容易在管道中增加阶段

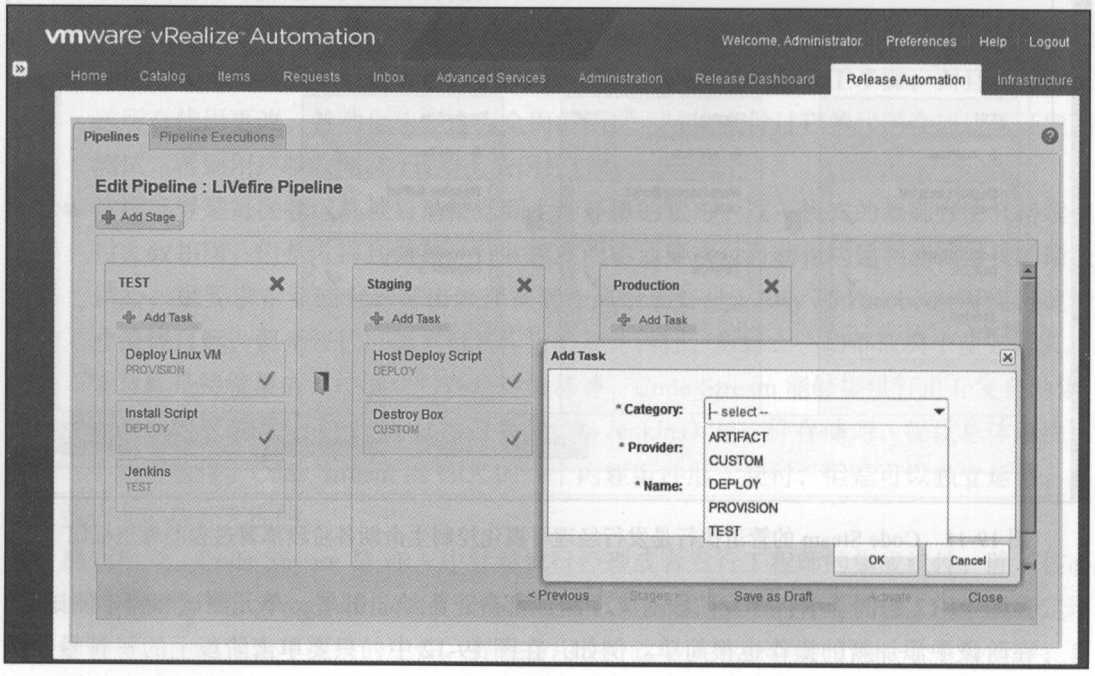


图 19-13 很容易通过 UI 在阶段中添加步骤

类似地，门控规则的创建是 UI 驱动的，与 vCenter Orchestrator（vCO）集成，vCO 也是 vRA 用具本身自带的（但是你可以自由地使用所需的 vCO 服务器）。这种门控功能（参见图 19-14）为用户提供了使用简单逻辑确定成败、通过 vRA 机制人工批准或者用 vCO 构建更复杂逻辑的灵活性。

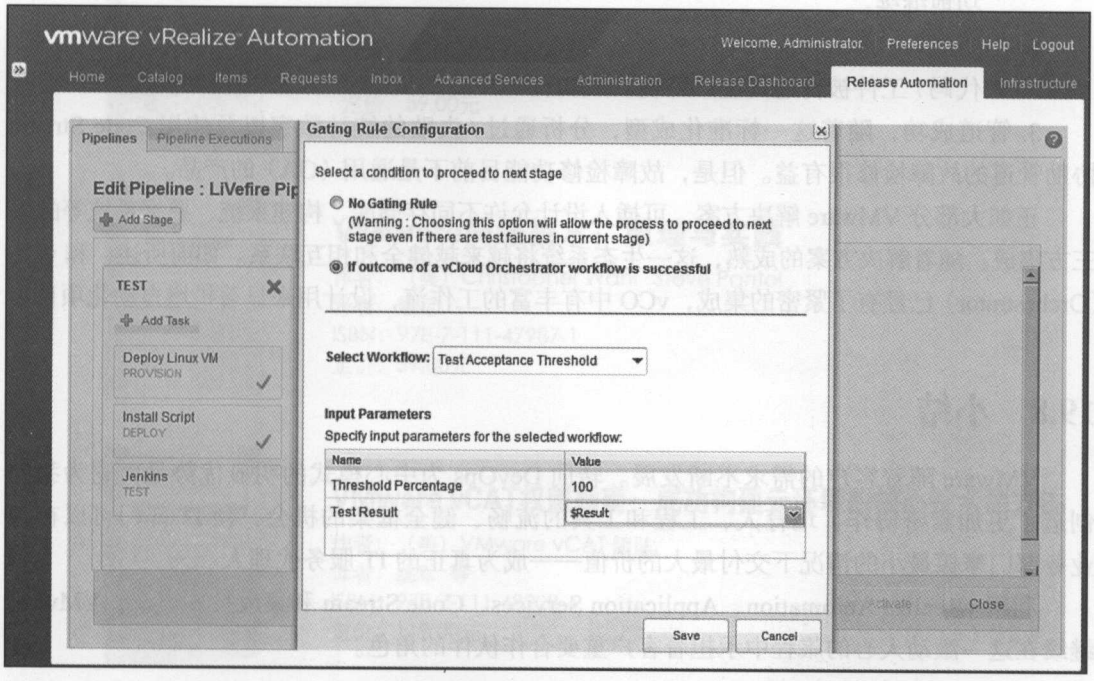


图 19-14　可以添加门控规则，管理阶段之间的推进。这些规则可以基于前一阶段的成功，基于 vCO 工作流、人工批准的结果或者这些因素与其他因素的组合

然后，Code Stream 管道可以根据对应的阶段初始化、构建、测试，经历开发、UAT 到预演阶段，同时由门控规则进行治理。这种治理确保过程的控制得到维持。在许多构建过程中可以看到 Release Automation（发行自动化）仪表盘，以识别关键里程碑、错误和成功。

我们将经历一个场景，确保对流程的理解：

1. 开发者在 Git 存储库中登入一些代码。此时，有许多方法可以转移到下一阶段，我们将专注于一个可能的路径。

2. Code Stream 中的管道执行被触发，有 3 个阶段：测试、预演和生产。

- 测试
  - VM 通过 vRA 目录自动部署。
  - Code Stream 以最新的代码在 VM 上执行构建，可能利用 Jenkins 自动化构建。
  - 通过 Selenium 等工具运行自动化测试。
  - 如果测试成功且通过门控规则，管道将继续推进到下一阶段。

- 预演
  - 因为可能是“褐地”部署，预演阶段可能不需要 VM。
  - 如果是这种情况，升级后的代码部署到现有 VM 作进一步测试。
  - 如果前述工作通过，测试阶段创建的 VM 将被摧毁，因为它不再有用。我们可以成功地继续。
- 生产
  - 代码 / 工件被传递到生产环境。成功！

3. 管道成功，随着这一标准化成型，分析通过 / 失败的统计数字以及使用 Code Stream 协助管道的故障检修很有益。但是，故障检修功能目前不是通用（GA）的产品。

正如大部分 VMware 解决方案，可插入设计允许不同存储库、构建系统、票据系统等的第三方集成。随着解决方案的成熟，这一生态系统将越来越健全和相互联系。如前所述，和 vCO（Orchestrator）已经有了紧密的集成，vCO 中有丰富的工作流，设计用来显著扩增自动化项目。

19.8 小结

VMware 随着客户的需求不断发展。转向 DevOps 为中心模式的明显优势是，它为我们创造了更加紧密协作、培育人、工程和工具的流畅、健全框架的机会，使 IT 部门可以在与业务部门摩擦最小的情况下交付最大的价值——成为真正的 IT 服务代理人。

利用 vRealize Automation、Application Services、Code Stream 和集成技术市场，VMware 继续在这一激动人心的旅程中承担着客户重要合作伙伴的角色。

参考文献

[1] http://www.melconway.com/Home/Conways_Law.html

[2] http://www.computerworld.com/article/2527153/it-management/opinion--the-unspoken-truth-about-managing-geeks.html
To be clear, the term *consumer* in IT does not necessarily apply to the end user. An IT department's main consumers may be developers.

[3] http://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=2068342

[4] You can obtain this script via VMware KB 2068342.

[5] Darwin-cli.jar needs to be downloaded from the Application Services appliance. You can find details at https://pubs.vmware.com/appdirector-1/index.jsp?topic=%2Fcom.vmware.appdirector.using.doc%2FGUID-D39A5F38-7EEF-43A3-8CF2-7ADA4C1E03F2.html.

[6] https://forge.puppetlabs.com/

[7] Note that Code Stream supports attaching to existing Artifactory servers as well.

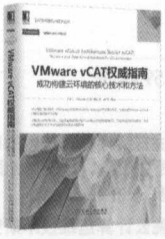
推荐阅读

VMware Virtual SAN权威指南

作者：（美）Cormac Hogan　Duncan Epping
译者：徐炯
ISBN：978-7-111-48023-5
定价：59.00元

VMware网络技术：原理与实践

作者：（美）Christopher Wahl　Steve Pantol
译者：姚军
ISBN：978-7-111-47987-1
定价：59.00元



VMware vCAT权威指南：成功构建云环境的核心技术和方法

作者：（美）VMware vCAT 团队
译者：姚军 等
ISBN：978-7-111-48228-4
定价：119.00元

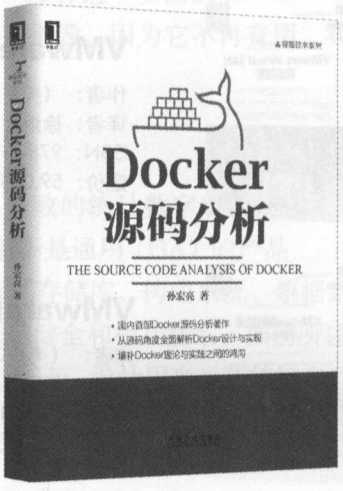
大规模Java平台虚拟化与调优

作者：（美）Emad Benjamin
译者：张卫滨 文建国
ISBN：978-7-111-49594-9
定价：59.00元

VMware站点恢复管理器管理实践

作者：（英）Mike Laverick
译者：马睿
ISBN：978-7-111-45735-0
定价：79.00元

推荐阅读



Docker 技术入门与实战

作者：杨保华 戴王剑 曹亚仑 ISBN：978-7-111-48852-1 定价：59.00元

本书作者之一杨保华博士在加入 IBM之后，一直从事云计算与软件定义网络领域的相关解决方案和核心技术的研发，热心关注 OpenStack、Docker 等开源社区，热衷使用开源技术，积极参与开源社区的讨论并提交代码。这使得他既能从宏观上准确把握 Docker 技术在整个云计算产业中的定位，又能从微观上清晰理解技术人员所渴望获知的核心之处。

—— 刘天成，IBM中国研究院云计算运维技术研究组经理

好的IT技术总是迅速“火爆”，Docker 就是这样。好像忽然之间，在企业一线工作的毕业生们都在谈论 Docker。在IT云化的今天，系统的规模和复杂性，呼唤着标准化的构件和自动化的管理，Docker 正是这种强烈需求的产物之一。这本书很及时，相信会成为IT工程师的宝典。

—— 李军，清华大学信息技术研究院院长

Docker源码分析

作者：孙宏亮 ISBN：978-7-111-51072-7 定价：59.00元

本书通过分析解读Docker源码，让读者了解 Docker的内部结构和实现，以便更好地使用Docker。该书的内容组织深入浅出，表述准确到位， 有大量流程图和代码片段帮助读者理解Docker各个功能模块的流程，是学习Docker开源系统的良师益友。

—— 寿黎旦，浙江大学计算机学院教授

这本书从源码的角度对Docker的实现原理进行了深入的探讨和细腻的讲解，将当前炙手可热的容器技术的背后机理讲解得如此的深入浅出和明白透彻。无论是Docker的使用者还是开发者，通过阅读此书都可以对Docker有更深刻的理解，能够更好地使用或者开发Docker。

—— 雷继棠，华为Docker Committer

作者简介

Trevor A. Roberts, Jr. VMware高级技术市场经理，拥有CCIE数据中心认证，是VMware数据中心设计和管理集中化认证高级专家。他因对IT社区的卓越贡献，曾被授予VMware vExpert、Cisco Data Center Champion和EMC Elect的称号。

Josh Atwell SolidFire的云架构师，专注于VMware和自动化解决方案。他是虚拟化社区的活跃分子，是CIPTUG、VMUG和UCS等技术用户组的领导人。

Egle Sigler 现为Rackspace的首席架构师。

Yvo van Doorn 是Chef解决方案架构师和新员工培训团队的负责人，有十多年的系统管理经验。

封面设计 锡 彬

DevOps for VMware Administrators

本书是第一本以结合VMware技术使用DevOps工具与实践为焦点的书籍。作者介绍了来自第三方和VMware自身的高价值工具，指导读者使用它们增强虚拟系统和应用程序的性能。读者将领略配置管理的自动化和优化、配给、日志管理、持续集成等技术过程。

本书还循序渐进地讲解利用Docker容器及Google Kubernetes大规模部署和管理应用程序的方法，并介绍了VMware最新的DevOps倡议，包括VMware vRealize Automation和VMware vRealize Code Stream。

通过阅读本书，你将学到：

- 理解DevOps工具和实践可以帮助VMware管理员解决的难题
- 使用Vagrant快速部署匹配生产系统规格的开发和测试环境
- 编写Chef“食谱”，合理化服务器配置和维护
- 用Ansible简化Unix/Linux配置管理和编排
- 采用Docker容器，实现更快速、更易行的应用程序管理
- 用Razor自动化全生命期配给
- 集成Microsoft PowerShell预期状态配置（DSC）和VMware PowerCLI，自动化关键Windows Server和vSphere VM管理任务
- 使用Puppet自动化基础设施配给、配置、编排和报告
- 用ELK（Elasticsearch、Logstash、Kibana）强化日志管理
- 用Git支持DevOps源代码管理，用Jenkins实现持续集成
- 用VMware vRealize Code Stream实现持续集成、交付和部署

PEARSON
www.pearson.com

vmware PRESS

PEARSON
ALWAYS LEARNING ALWAYS LEARNING ALWAYS LEARNING

上架指导：计算机\虚拟化
ISBN 978-7-111-52478-6
9 787111 524786 >
定价：69.00元

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn